6.005 Software Construction

Objectives

The objectives of this course are that, by the end:

1. You will be able to articulate and apply principles of engineering sensibility: simplicity, safety from bugs, ease of understanding, readiness for change.
2. You will have a solid grasp of, and ability to apply, key software engineering ideas, including interfaces, representation invariance, specifications, invariants, data abstraction, design patterns, and unit testing.
3. You will be able to design, implement, and test a small- to medium-scale software system (thousands of lines of code, multiple modules).
4. You will have experience developing software collaboratively in a team.
5. You will have experience with modern programming tools (e.g. Eclipse, Subversion, JUnit) and modern programming technologies (e.g. I/O, regular expressions, network sockets, threads, GUIs)

Course Elements

Lectures. There are two 1-hour lectures each week. You are expected to attend all the lectures and to participate actively in discussions.

Recitations. There are two 1-hour recitations each week. You are expected to attend the recitation that was assigned to you by the Registrar. You are expected to attend all recitations and participate actively. Recitation exercises will be collected and graded for participation.

Laptops required. Recitations (and occasional lectures) will have programming exercises that require a laptop. If you don't have your own laptop and you need to borrow one, IS&T has a laptop loaner program that will lend you one.

Text. There is no course text, but lecture and recitation notes will be posted online.

Problem sets. To consolidate your understanding of the lecture ideas, you will do eight problem sets, involving both design and implementation work. Problem sets will be done individually.

Code review. After each problem set submission deadline, there will be a 3-day code-reviewing period when other students, staff, and alumni will give you feedback about the code you submitted, using a web-based system. You will be expected to participate in this process by reviewing some of your classmates' code. More details about objectives and guidelines for the code reviewing process will be found in a separate document on this web site.

Projects. You will complete two small software development projects, each of which goes all the way from specification and design to coding and testing. All projects will be done in teams of three students. You'll have the opportunity to choose whom to work with, and we'll do our best to honor your choices. Any pair of students, however, will only be allowed to work together on a single project. Each team member is required to participate roughly equally in every activity (design, implementation, test, documentation), and we may ask for an accounting of what each team member did. Each project will have a deliverable after one week (usually an initial design and some initial code), and a final deliverable on the deadline. A single grade will be assigned to all students in the project.

Team meetings. During the projects, you and your project team will meet with your TA to discuss the work. Your TA will assign a grade based in part on this meeting. Team meetings will usually be scheduled during lecture or recitation times that will be reserved for this purpose.

Quizzes. There will be two quizzes, on dates specified on the course calendar. Each quiz will be comprehensive, drawing on any lecture and recitation topics covered up to that point in the course, so Quiz 2 may include topics that were already covered on Quiz 1.

Grading

The relative contributions of the various elements to your grade are:

- **Quizzes: 20%.** Q1 and Q2 have equal weight.
- **Problem sets: 40%.** PS0 is worth half as much as PS1-PS7.
- **Projects: 30%.** Project 1 is worth 12%, and Project 2 is worth 18%.
- **Participation: 10%.** The participation grade will be determined by your participation in recitations, code reviewing, lectures, and the online Piazza forum (roughly in that order of precedence).

Lateness and Extensions

To give you some flexibility for periods of heavy workload, minor illness, absence from campus, and other unusual circumstances, you may request limited extensions on problem set deadlines, called *slack days*. Each slack day is a 24-hour extension on the deadline. You have a budget of **5 slack days** for the entire semester, which you may apply to any combination of problem sets. You can use at most 3 slack days for a given problem set; after three days, whatever is in your Subversion repository is what we will grade.

You must request your extension before the problem set is due, by logging into the code-reviewing system and clicking on Request Extension.

The system keeps track of your slack days and informs you how many you have left.

The grading process grades what you've submitted by your extended deadline, but the code reviewing process pays no attention to your extension. We will ask people to review whatever code is in your Subversion repository at the *original* deadline. So even if you request an extension, you should commit as much as you can before the original deadline, so that your code reviewers have something to look at. But you yourself must wait to review other people's code until *after* your extended deadline. Looking at other students' code before handing in an individual problem set is a violation of the course collaboration policy (below).

Slack days are atomic; you can't chop them up into slack hours or minutes. If you choose to use a slack day because you're struggling with an assignment, GO TO BED AND SLEEP, and get help in the morning. Seriously, that's why the smallest unit is a day.

Slack days apply only to individual problem sets, not to team projects.

If you have used up your slack days, or exceeded the 3-day limit for a single assignment, you will need a lecturer's permission and support from an S^3 dean for more extension.

Collaboration

In line with the Departmental Guidelines Relating to Academic Honesty, here are our expectations regarding collaboration and sharing.

For the team projects, you are encouraged to collaborate with your partners on all aspects of the work, and each of you is expected to contribute a roughly equal share to design and implementation. You may reuse designs, ideas and code from your own work earlier in the semester (even if it was done in a team project with a different partner). You may also use material from external sources, so long as: (1) the material is available to all students in the class; (2) you give proper attribution; and (3) the assignment itself allows it. In particular, if the assignment says "implement X," then you must create your own X, not reuse someone else's.

Problem sets are intended to be primarily individual efforts. You are encouraged to discuss approaches with other students but your write-up must be your own. You should not make use of any written solutions or partial solutions produced by others. Material from external sources can also be used with proper attribution, but only if the assignment allows it. You may not use materials produced as course work by other students in the course, whether in this term or previous terms, nor may you provide work for other students to use.

Copying work, or knowingly making work available for copying, in contravention of this policy is a serious offense that may incur reduced grades, failing the course and disciplinary action.

Questions?

Before you ask
... try to find the answer yourself, using the course Stellar site, or the Java documentation, or the course Piazza forum, or just by searching the Web.

For problem set questions and debugging help
...ask an LA using the Karga online queue
...or ask on the course Piazza forum
...or ask your TA.

For group project questions
... ask your TA mentor

For questions about lecture content, to give feedback, and for special requests and concerns
...ask a lecturer.

6.005[4] Static Checking

9/~~0~~7

~~Teach~~ Prof Rob Miller

L1 Static checking
- Java
- static typing
- hacking vs engineering

PS 0 due tmo midnight - worth half credit
- to review policies + practice submitting

peer Code reviewing Fri + Sat - new this year
- not in grade - others reviews
- your reviews in grade

PS1 out Fri, due next thur

Hailstone Sequence

$n = 3$

generate more by
- if $n = 1 \rightarrow$ stop
- if n = even
  - $n = n/2$
- else n = odd
  - $n = 3n + 1$

So 3, 10, 5, 16, 8, 4, 2, 1

or 7, 22, 11, 34, 17, etc

not clear when it comes back down, if ever

Control statements similar to python

- Syntatic differences
- Eclipse has format command

have to declare variables in Java

+ give types

<u>int</u> n = 3;

(def) <u>type</u> = set of values + operations can build on type

<u>int</u> 0, 1, -2

+ : int x int → int

/ : int x int → int

So 3/2 = 1 (truncates fraction)

<u>double</u> 0.5, 1.333

floating pt # - all real #

+ : d x d → d

/ : d x d → d

3

Operation (+) called (overloaded)
- depends/differs if int or double

String "abc" +: string x string → string
- is in a library actually

examples

+: i x d → d

+: s x i → s

Eclipse will tell us errors on type

Things checked at compile time vs run time
- Checked when compile
- or immediatly in Eclipse on sidebar w/ icons
So can fix issues before Runtime

(4)

System.out.println(a);

Array of ints, int[] a

↑ space

ints are pointers / primitives

int n = 3

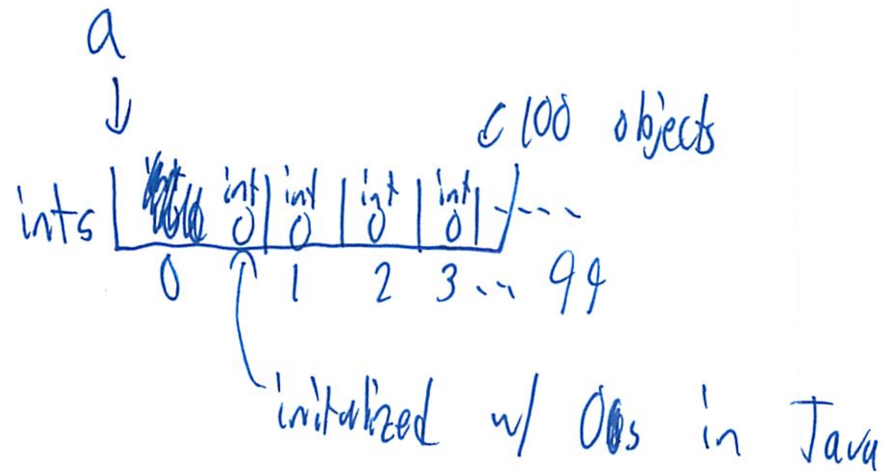
n → 3

n = 4

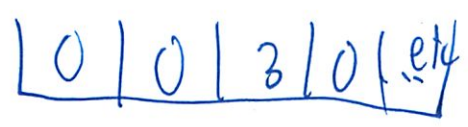
n ~~→~~ 3
↳ 4

for example,

int[] a = new int[100];



Can reprint - a[2] = 3



Can do a.length
↳ 100

a = new int [4]

now a ✱→ | 0 | 0 | 3 | etc |

↓

| 0 | 0 | 0 | 0 |

← floats in space till Java auto garbage collects

You have to import arrays 1st

At top

import Java.util.Arrays;

CTRL + [Space] is auto complete

Or in Eclipse Sorce → Organize Imports and it will auto do

(6). Your code needs to be ready for obvious changes

Such as starting n = 83

exceeds the 100 spaces in array

So you get an Exception (at runtime)

(in C, C++ would get a buffer overflow)

Will use lists instead

List <Integer> l = new ArrayList<Integer>();

need to say this (^ under Integer)

not int

Object value, not primitive value

List is abstract type

- just set of operations
- not values
- needs concrete implementation like ArrayList

~~St~~ starts empty

So l.add(n) is like append

Writing methods (functions)

- public → anyone can use
- static → no self parameter
  implicit
  called "this"
  can say max() not Object.max()
- need to write Javadoc comment
  @param
  @return

For loops

- similar to PHP for each

```
for (int x : l)
```

x = each value, l = the list

~~Does not work 2 a~~

Watch hidden assumptions/boundary condition
- either fix
- or say Requires in @param
- negitive values?
- list empty?

(8)

Can also do standard for loop

```
for (int m= 1; m< 1000; m++)
```

113383 ← why did it crash here

L1: Static Checking

Today

- Static typing
- Snapshot diagrams
- Hacking vs. engineering

Required reading (from the Java Tutorial)

- Language Basics
- Numbers and Strings
- Defining methods
- Calling methods
- Returning a value from a method
- Hello World!

Hailstone Sequence

As a running example, we're going to explore the hailstone sequence, which is defined as follows. Starting with a number n, the next number in the sequence is n/2 if n is even, or 3n+1 if n is odd. The sequence ends when it reaches 1. Here are some examples:

2, 1

3, 10, 5, 16, 8, 4, 2, 1

4, 2, 1

2^n, 2^{n-1} , ... , 4, 2, 1

5, 16, 8, 4, 2, 1

7, 22, 11, 34, 17, 52, 6, 13, 40, ...? (not clear where this stops!)

Because of the odd-number rule, the sequence may bounce up and down before decreasing to 1. (It's conjectured that the hailstone sequence reaches 1 for all starting *n*, but that's still an open question.

Why is it called a hailstone sequence? Because hailstones form in clouds by bouncing up and down, until they eventually build enough weight to fall to earth.

Computing Hailstones

Here's Python code for computing the hailstone sequence and print it:

```
n = 3
while n != 1:
    print n
```

```
        if n % 2 == 0:
            n = n / 2
        else:
            n = 3 * n + 1
    print n
```

Here's the equivalent Java code:

```
    int n = 3;
    while (n != 1) {
        System.out.println(n);
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    System.out.println(n);
```

A few things are worth noting here. The basic semantics of expressions and statements in Java are very similar to Python: while and if behave the same, for example. Note the differences in syntax, however. Java requires semicolons at the ends of statements, parentheses around the conditions of the if and while, and curly braces around blocks, instead of indentation. But you should still indent the block, even though Java won't pay any attention to your extra spaces. Programming is a form of communication, and you're communicating not only to the compiler, but to human beings. Humans need that indentation. We'll come back to this later.

Static Typing

The most important semantic difference between the Python and Java code above is the declaration of the variable n, which specifies its type: int.

A type is a set of values, along with operations that can be performed on those values. Java has several primitive types, among them:

- int (for integers like 5 and -200, but limited to the range ± 2 billion)
- long (for larger integers up to 2^{63})
- boolean (for true or false)
- double for floating-point numbers (which represent a subset of the real numbers)
- char (for characters like 'A' and '$')

Java also has object types. String represents a sequence of characters, like a Python string. BigInteger represents an integer of arbitrary size, so it acts like a Python number.

Operations are functions that take inputs and produce outputs (and sometimes change the values themselves). The syntax for operations varies, but we still think of them as functions:

a + b	+ : int x int -> int
bigint1.add(bigint2)	add: BigInt x BigInt -> BigInt
str.length()	length: String -> int

Contrast Java's string length with Python's len(s). It's the same operation, just written with a different syntax.

Some operations are *overloaded* in the sense that the same operation name is used for different types. The arithmetic operators +, -, *, / are heavily overloaded for the numeric primitive types in Java.

Method definitions can also be overloaded. Most programming languages have some degree of overloading.

Java is a statically-typed language. The types of all variables are known at compile time (before the program runs), and the compiler can therefore deduce the types of all expressions as well. If a and b are declared as ints, then the compiler concludes that a+b is also an int. The Eclipse environment does this while you're writing the code, in fact, so you find out about many errors.

In dynamically-typed languages like Python, this kind of checking is deferred until runtime (while the program is running).

Bugs are the bane of programming. Many of the ideas in this course are aimed at eliminating bugs from your code, and static checking is the first idea that we've seen for this. Static checking prevents a large class of bugs from infecting your program: to be precise, bugs caused by applying an operation to the wrong types of arguments. If you write a broken line of code like:

```
"5" * "6"
```

then static checking will catch this error while you're still programming, rather than waiting until this line is reached during execution of the code.

It's useful to think about three kinds of automatic checking that a language can provide:

- Static checking: the bug is found (automatically) before you run.
- Dynamic checking: the bug is discovered
- No checking: the language doesn't help you find the error at all.

Let's try some examples of buggy code and see how they behave in Java. Are they caught statically, dynamically, or not at all?

```
int n = 5;
if (n == true) { }

int n = 2000000000; // 2,000,000,000
n = n * 2;

double d = 1 / 5;

int n = 5;
n = n/0;

double d = 1;
d = d/0;
```

Arrays and Collections

Let's change our hailstone computation so that it stores the sequence in a data structure, instead of just printing it out. Java has two kinds of list-like types that we could use: arrays and Lists.

Arrays are fixed-length sequences of another type *T*. For example, here's how to declare an array variable and construct an array value to assign to it:

```
int[] a = new int[100];
```

The int[] array type includes all possible array values, but a particular array value, once created, can never change its length. Operations on array types include indexing (a[2]), assignment (a[2]=0), and getting the length (a.length).

Here's a crack at the hailstone code using an array. We start by constructing the array, and then use an index variable *i* to step through the array, storing values of the sequence as we generate them.

```
int[] a = new int[100];
int i = 0;
int n = 3;
while (n != 1) {
    a[i] = n;
    i++;  // very common shorthand for i=i+1
    if (n % 2 == 0) {
        n = n / 2;
    } else {
        n = 3 * n + 1;
    }
}
a[i] = n;
i++;
```

Something should immediately smell wrong in this approach. What's that magic number 100? What would happen if we tried an *n* with a very long hailstone sequence? We have a bug. Would Java catch the bug statically, dynamically, or not at all? Incidentally, bugs like these -- overflowing a fixed-length array, which are commonly used in less-safe languages like C and C++ that don't do automatic runtime checking of array accesses – have been responsible for a large number of network security breaches and internet worms.

Instead of a fixed-length array, let's use the List type. Lists are variable-length sequences of another type *T*. They are declared like List<Integer>, constructed like new ArrayList<Integer>(), and the operations include indexing (list.get()), assignment (list.set()), and length (list.size()).

Note that List is an interface, a type that can't be constructed directly, but simply specifies the operations that a List must provide. We'll talk about this notion in a future lecture on abstract data types. ArrayList is a class, a concrete type that provides implementations of those operations. ArrayList isn't the only implementation of the List type, though it's the most commonly used one. LinkedList is another. Check them out in the Java API documentation.

Note also that we unfortunately can't write List<int> in direct analog to int[]. Lists only know how to deal with object types, not primitive types. In Java, each of the primitive types (lowercase) has an equivalent object type (capitalized) that we have to use in the angle brackets. In other respects, Java automatically converts between int and Integer, so we can write Integer i = 5 without any type error.

Here's the hailstone code written with list:

```
List<Integer> l = new ArrayList<Integer>();
int n = 3;
while (n != 1) {
    l.add(n);
    if (n % 2 == 0) {
        n = n / 2;
    } else {
        n = 3 * n + 1;
    }
}
l.add(n);
```

Iterating

A for loop steps through the elements of an array or a list, just as in Python, though the syntax looks a little different. For example:

```
int max = 0;
for (int x : l) {
        max = Math.max(x, max);
}
```

The same code would work if the list *l* were replaced by an array *a*.

Methods

In Java, statements generally have to be inside a method, and every method has to be in a class, so the simplest way to write our hailstone program looks like this:

```
public class Hailstone {
        /**
         * Compute a hailstone sequence.
         * @param n  Starting number for sequence.  Assumes n > 0.
         * @return hailstone sequence starting with n and ending with 1.
         */
        public static List<Integer> hailstoneSequence(int n) {
                List<Integer> l = new ArrayList<Integer>();
                while (n != 1) {
                    l.add(n);
                    if (n % 2 == 0) {
                        n = n / 2;
                    } else {
                        n = 3 * n + 1;
                    }
                }
                l.add(n);
                return l;
        }
}
```

public means that any code, anywhere in your program, can refer to the class. Other access modifiers (like *private*) are used to get more safety in a program, and to guarantee immutability for immutable types. We'll talk more about them in an upcoming lecture.

static means that the method doesn't take a self parameter (which in Java is implicit), and it can't be called on an object (unlike the List add() method or the String length() method, for example, which require an object to come first). Instead, the right way to call this method uses the class name:

```
Hailstone.hailstoneSequence(83)
```

Take note also of the comment before the method, because it's very important. This comment is a *specification* of the method, describing the inputs and outputs of the operation. The specification should be concise and clear and precise. The comment provides information that is *not* already clear from the method types. It doesn't say, for example, that n is an integer, because the int n declaration just below already says that; but it does say that n must be a nonnegative integer, which is an important assumption for the caller to know.

We'll have a lot more to say about how to write good specifications in a few lectures, but you'll have to start reading them and using them right away.

Snapshot Diagrams

It will be useful for us to draw pictures of what's happening at runtime, in order to understand subtle issues. Refer to the Lecture 1 Addendum on Stellar for more about the syntax of snapshot diagrams.

Mutating Values vs. Changing Variables

Snapshot diagrams give us a way to visualize the distinction between changing a variable and changing a value. When you assign to a variable, you're changing where the variable's arrow points. You can point it to a different value.

When you assign to the contents of a mutable value – such as an array or list – you're changing references inside that value.

Change is a necessary evil. Good programmers avoid things that change, because they may change unexpectedly.

Immutability (immunity from change) is a major design principle in this course. Immutable types are types whose values can never change once they have been created. (At least not in a way that's visible to the outside world – there are some subtleties there that we'll talk more about that in a future lecture about immutability.) Which of the types we've discussed so far are immutable, and which are mutable?

Java also gives us immutable references: variables that are assigned once and never reassigned. To make a reference immutable, declare it with the keyword *final*:

```
final int n = 5;
```

If the Java compiler isn't convinced that your final variable will only be assigned once at runtime, then it will produce a compiler error. So final gives you static checking for immutable references.

It's good practice to use final for declaring the parameters of a method and as many local variables as possible. Like the type of the variable, these declarations are important documentation, useful to the reader of the code and statically checked by the compiler.

There are two variables in our hailstoneSequence method: can we declare them final, or not?

```
public static List<Integer> hailstoneSequence(final int n) {
    final List<Integer> l = new ArrayList<Integer>();
```

Documenting Assumptions

Writing the type of a variable down documents an assumption about it: e.g., *this variable will always refer to an integer.* Java actually checks this assumption at compile time, and guarantees that there's no place in your program where you violated this assumption.

Declaring a variable final is also a form of documentation, a claim that the variable will never change after its initial assignment. Java checks that too, statically.

We documented another assumption that Java doesn't check: that n must be nonnegative.

Why do we need to write down our assumptions? Becaue programming is full of them, and if we don't write them down, we won't remember them, and other people who need to read or change our programs later won't know them. They'll have to guess.

Programs have to be written with two goals in mind: communicating with the computer (i.e., first persuading the compiler that your program is sensible, and then computing the right result at runtime). But communicating with other people.

Hacking vs. Engineering

We've written some hacky code in this lecture. Hacking is often marked by unbridled optimism:

- writing lots of code before testing any of it
- keeping all the details in your head, assuming you'll remember them forever, instead of writing them down in your code
- assuming that bugs will be nonexistent or else easy to find and fix

But software engineering is not hacking. Engineers are pessimists:

- write a little bit at a time, testing as you go. In the next lecture, we'll talk about test-first programming.
- document the assumptions that your code depends on
- defend your code against stupidity – especially your own! Static checking helps with that.

Our primary goal in this course is learning how to produce software that is:

- **Safe from bugs**. Correctness (correct behavior right now), and defensiveness (correct behavior in the future).
- **Easy to understand**. Has to communicate to future programmers who need to understand it and make changes in it (fixing bugs or adding new features). That future programmer might be *you*, months or years from now. You'll be surprised how much you forget if you don't right it down, and how much it helps your own future self to have a good design.
- **Ready for change**. Software always changes. Some designs make it easy to make changes; others require throwing away and rewriting a lot of code.

These are the Big Three. It's worth considering every language feature, every programming practice, every design pattern that we study in this course, and understanding how they relate to the Big Three.

The main idea we introduced today is static checking. It helps with safety by catching type errors before runtime. It also helps with understanding, because types are explicitly stated in the code.

Why we use Java in this course

Since you've had 6.01, we're assuming that you're comfortable with Python. So why aren't we using Python in this course? Why do we use Java in 6.005?

Safety is the first reason. Java has static checking (primarily type checking). We're studying software engineering in this course, and safety from bugs is a key tenet of that approach. Java dials safety up to 11, which makes it a good language for learning about good software engineering practices. It's certainly possible to write safe code in dynamic languages like Python, but it's easier to understand what you need to do if you learn a safe language first.

Ubiquity is another reason. Java is widely used in research, education, and industry. Java runs on many platforms, not just Windows/Mac/Linux. Java can be used for web programming (both on the server and in the client), and native Android programming is done in Java. Although other programming languages are far better suited to teaching programming (Scheme and ML come to mind), regrettably these languages aren't as widespread in the real world. Java on your resume will be

recognized as a marketable skill. But don't get us wrong: *the real skills you'll get from this course are not Java-specific, but carry over to any language that you might program in.* The most important lessons from this course will survive language fads: safety, clarity, abstraction, engineering instincts.

In any case, a good programmer *must* be **multilingual.** Programming languages are tools, and you have to use the right tool for the job. You will likely have to pick up other programming languages before you even finish your MIT career (Javascript, C/C++, Scheme or Ruby or ML or Haskell), so we're getting started now by learning a second one.

As a result of its ubiquity, Java has a wide array of interesting and useful **libraries** (both its enormous built-in library, and other libraries out on the net), and excellent free **tools** for development (IDEs like Eclipse, editors, compilers, virtual machines, test frameworks, profilers, code coverage). Even Python is still behind Java in the richness of its ecosystem.

There are some reasons to regret using Java. It's wordy, which makes it hard to write examples on the board. It's large, internally inconsistent (e.g. *final* means several different things), and weighted with the baggage of old languages like C/C++ (the primitive types and the *switch* statement are good examples). It's large, having accumulated many features over the years. It has no interpreter (like Python's) where you can play with small bits of code. It has no lambda expressions, continuations, or tail recursion, so functional programming is harder and less interesting in Java.

Books about Java

There are many good books on Java, and a few good web sites. When we use new bits of Java in lecture, the lecture notes will include links to a free web resource called *The Java Tutorial* published by Oracle. When the Java Tutorial does not provide sufficient detail, we direct you to sections in other books.

Here are some other good books about Java:

- Effective Java, by Bloch. This book contains itemized advice for Java programmers on several issues. The importance of many of these issues transcends Java. The book contains fantastic insights, directing you to effectively use features of the language and its libraries to develop good Java programs. It is not available online, but **we strongly recommend you read this book for 6.005 and use it as a reference.**
- The Java Programming Language, 4th Edition, by Arnold, Gosling, and Holmes. This is the definitive source on Java. It explains (and justifies) every feature of the language. It is useful if you want to understand a tricky issue, but usually less useful than other Java references (because it is very detailed and presents all the features of the language in detail, when some of them are not very useful). It is not available on-line.
- Java in a Nutshell, 5th Edition, by Flanagan. This is a practical introduction to Java. It is not as deep, but is useful for quickly gaining skills. It is roughly equivalent to the Java Tutorial, and is available online through Safari, an on-line resource available through the MIT libraries.
- Many other Java books are available through Safari and Books 24x7, another on-line collection of books available through the MIT libraries.

A Guide to Snapshot Diagrams

Purpose

Snapshot diagrams represent the internal state of a program at runtime – its stack (methods in progress and their local variables) and its heap (objects that currently exist).

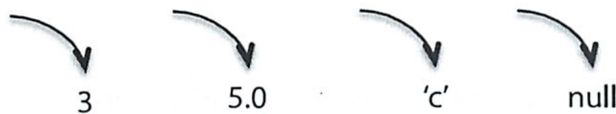
Here's why we use snapshot diagrams in 6.005:

- To talk to each other through pictures (in lecture, recitation, and team meetings)
- To illustrate concepts like primitive types vs. object types, immutable values vs. immutable references, pointer aliasing, stack vs. heap, abstractions vs. concrete representations.
- To help explain your design for your team project (to each other and to your TA)
- To pave the way for richer design notations in subsequent courses. Snapshot diagrams generalize into object models in 6.170.

Although the diagrams below use examples from Java, the notation can be applied to most object-oriented programming languages, e.g. Python, Javascript, C++, Ruby.

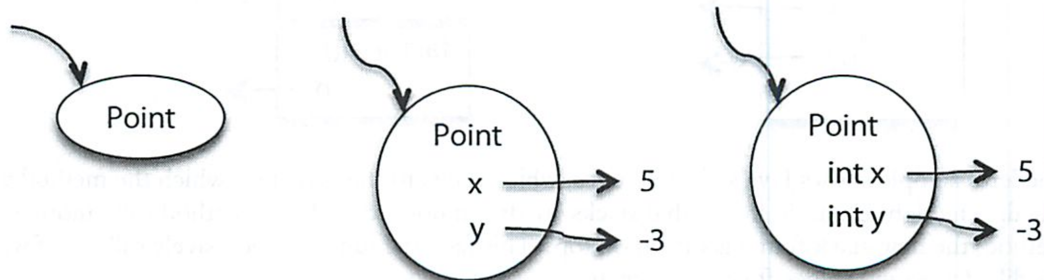
Primitive values

Primitive values are represented by bare constants. (The incoming arrow is a reference from a variable or an object field.)



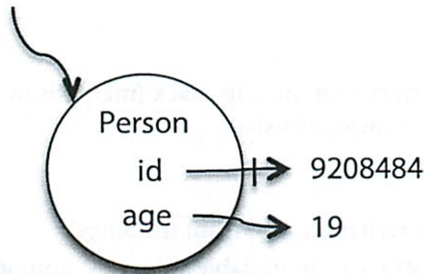
Object values

An object value is a circle labeled by its type. When we want to show more detail, we write field names inside it, with arrows pointing out to their values. For still more detail, the fields can include their declared types. Some people prefer to write *x:int* instead of *int x*; both are fine.



Immutability

Immutable references (called *final* in Java) are denoted by a crossbar just behind the arrowhead. Here's an object whose *id* never changes (it can't be reassigned to a different number), but whose *age* can change.

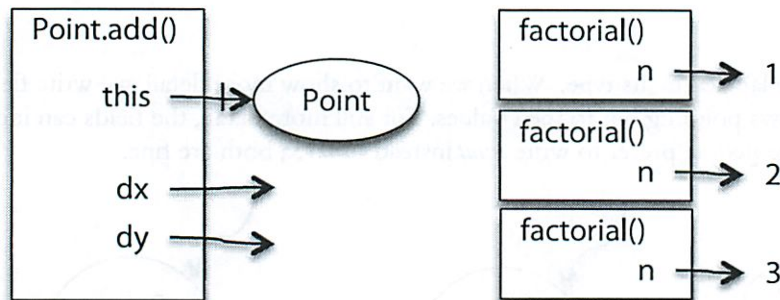


Immutable objects (intended by their designer to always represent the same value) are denoted by a double border. For example, here's an Integer object, an object value that represents the "boxed" form of an int:

Note that we also omitted the field name here, because it isn't necessary to understand the picture.

Stack frames

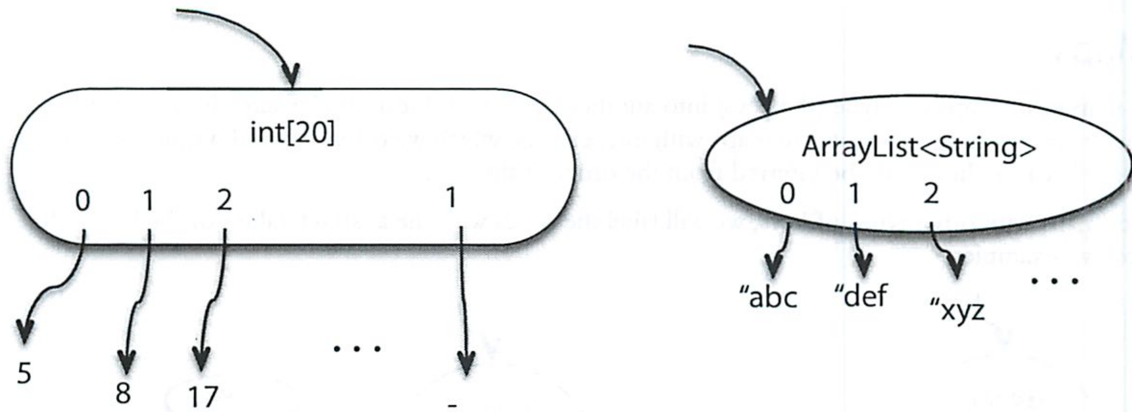
A stack frame is a method in progress. It is labeled by the method name (including its class name or parameter types if necessary to be clear), and contains references for parameters and local variables. Examples:



The left example shows Java's *this* reference, which refers to the object on which the method was called. The right example shows that stacks are drawn upwards: when a method calls another method, the new stack frame is drawn on top. This factorial function recursively calls itself with steadily decreasing values for the parameter *n*.

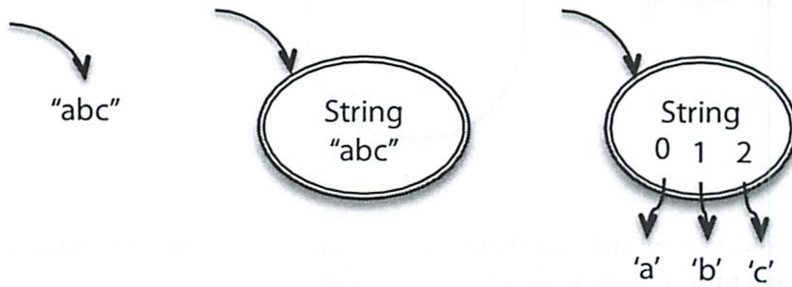
Arrays and Lists

Like other object values, arrays and lists are labeled with their type. In lieu of field names, we label the outgoing edges with indexes (0, 1, ...). When the sequence of elements is obvious, we may omit the index labels.

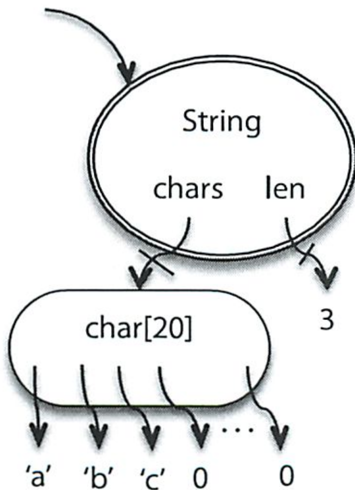


Strings

Here are several ways to write strings that emphasize different things. The leftmost treats a string as a primitive, which we'll often do simply because strings are immutable and frequent, so it's convenient. The middle picture treats a string as an object value, but shows its abstract value. The right picture regards a string as a sequence of characters.



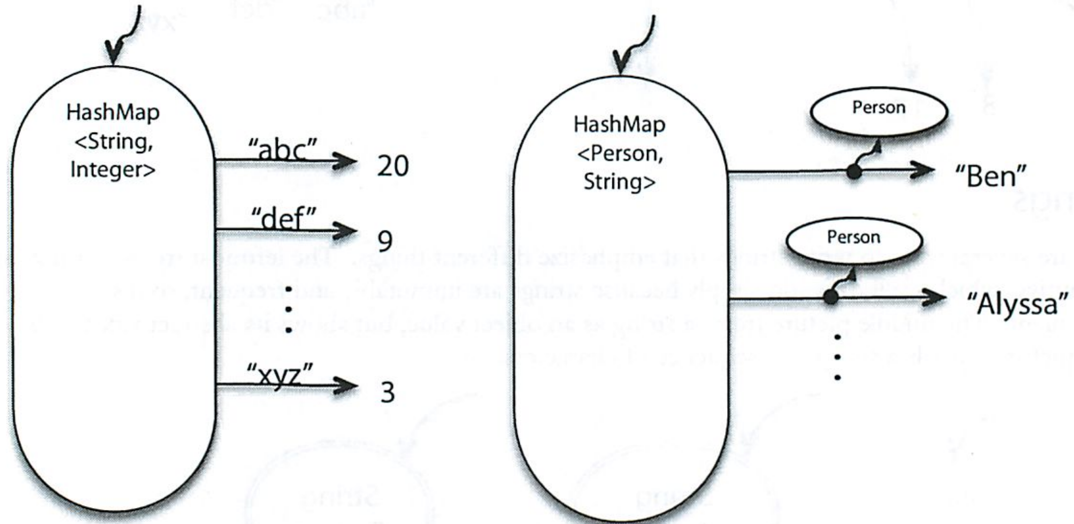
Finally, when we're talking about the concrete internal representation of a string in Java, we might draw a picture like this, which shows the string's character array:



Maps

A map value maps one type (the keys) into another type (the values). We've already seen examples of these: arrays, strings and lists are maps with integer keys, which we often omitted writing on the edges because they could be inferred from the order of the edges.

For maps with other types of keys, we will label the edges with the abstract values of the keys. Here are two examples:



This example uses some abbreviation: strings and integers are drawn as their abstract primitive values, even though Java internally represents them as object values.

No blue pen

6.005 Recitation

Will code in recitation
- optional

Term Review

- int
- static - belongs to class, not instance
- over loading

Reviewing errors from yesterdays

Since Java is picky about types
Need to be careful to convert

int i = Integer . parse Int (s) / ;

Also be sure to save
or "not valid assignment operator"

Remember int gets truncated

②

shortkey.com/dice

Null pointer exception !f null

* make sure always correct / or
  - like ⊖ or null *

always something weird in Java

Need to learn syntax

(Recitation going too fast - partner programming :/)

== vs !=

Convert between int / Integer / double

6.005 Doing PS0

JUnit under debug
- actually kinda cool
- best for reusable, core components

(Rushing - not cocentrating)

Oh from their website - read!

?. Behavior unspecifed means don't build test case

Run debug on the <u>test file</u>

Error messages actually good

Should return latest possible

Void = method does not return a new

? oh that is what it returns

ob seperate

Rushed through in 55 min

6.005 Software Construction
Problem Set 0: Getting Started
Due: Thursday, September 8, 2011. 11:59pm

Before you start coding

- **Install the software you'll need on your laptop.** You need to install three things for this problem set:
  - **JDK 6** (for Windows or Linux; Mac users probably already have Java, if not, get it here for 10.5. If you have a PowerPC or early Intel Mac or are running OS X 10.4, come talk to us). From this web page, download "JDK 6 Update 27"; you don't need NetBeans or Java EE.
  - **Eclipse 3.7**. Choose "Eclipse IDE for Java Developers", which will download a ZIP file. Unpack the ZIP file, go inside the resulting folder, and run Eclipse.
  - **Subclipse**. To install Subclipse, use Eclipse's Software Updates mechanism. The installation instructions on the Subclipse website are out of date, so follow these instructions instead.

Athena already has this software installed, so if you'll be using an Athena, you don't need to install anything.

Eclipse

The Eclipse integrated development environment (IDE) is a powerful, flexible, complicated, and occasionally frustrating set of tools for writing, modifying, and debugging programs. It is especially useful for working in Java.

On Athena, Eclipse is in the `eclipse-sdk` locker; the command name is `eclipse`.

When you run Eclipse, you will be prompted for a "workspace," which is where Eclipse will store all the different projects you work on. On Athena, for example, the default location is a directory called `workspace` in your home directory. In addition to code, Eclipse stores its own "metadata" in hidden folders in the workspace. **You should not run more than one copy of Eclipse at the same time** with the same workspace, or the metadata will become corrupted.

The first time you run Eclipse, it will show you a welcome screen. Click the button to go directly to the "workbench" and you're ready to begin. For some useful tips and troubleshooting information, you can look at the 6.170 Eclipse Reference, but disregard the 6.170-specific notes there.

Tab policy

At this time you should also go through and change tabs to spaces. Go to Window (Eclipse if you're running a Mac) → Preferences... →. In the toolbar go to Java → Code Style → Formatter. Click the "Edit..." button next to the active profile. In the new window you should change the Tab policy to " Spaces only " Keep the Indentation size and Tab size at 4. Enter a new name for the policy and press ok.

Subversion

To turn in this problem set, you will commit your code to Subversion. The last committed code before the problem set duedate will be used for grading.

O'Reilly publishes *Version Control with Subversion*, conveniently available online. This book describes what Subversion (or SVN) is and how to use it:

> Subversion is a free/open-source version control system. That is, Subversion manages files and directories, and the changes made to them, over time. This allows you to recover older versions of

> your data, or examine the history of how your data changed. In this regard, many people think of a version control system as a sort of "time machine".
>
> Subversion can operate across networks, which allows it to be used by people on different computers. At some level, the ability for various people to modify and manage the same set of data from their respective locations fosters collaboration. Progress can occur more quickly without a single conduit through which all modifications must occur. And because the work is versioned, you need not fear that quality is the trade-off for losing that conduit—if some incorrect change is made to the data, just undo that change. [What is Subversion?]

If you have used other version control software, such as CVS, many of the concepts and procedures of SVN will be familiar to you. If not, there are two important ideas to learn: *repositories* and *working copies*.

Repositories

> Subversion is a centralized system for sharing information. At its core is a repository, which is a central store of data. The repository stores information in the form of a filesystem tree—a typical hierarchy of files and directories. Any number of clients connect to the repository, and then read or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others.
>
> [...] What makes the Subversion repository special is that it remembers every change ever written to it: every change to every file, and even changes to the directory tree itself, such as the addition, deletion, and rearrangement of files and directories.
>
> When a client reads data from the repository, it normally sees only the latest version of the filesystem tree. But the client also has the ability to view previous states of the filesystem. For example, a client can ask historical questions like, "What did this directory contain last Wednesday?" or "Who was the last person to change this file, and what changes did he make?" [The Repository]

Every SVN repository has a URL at which it can be accessed.

Working Copies

In order to make changes to files in the repository, you must obtain (or "check out") a copy of the current version of those files:

> [... E]ach user's client contacts the project repository and creates a personal working copy—a local reflection of the repository's files and directories. Users then work simultaneously and independently, modifying their private copies. Finally, the private copies are merged together into a new, final version. [Versioning Models]

Any number of people can have any number of working copies (or "checkouts") of different parts of a single SVN repository. Those working copies might be on different machines and have different versions of files.

SVN in 6.005

In 6.005 you will be using a Subversion repository. The repository is located at https://svn.csail.mit.edu /6.005/fa11. You can also access it from any web browser and see the latest version of any file in the repository—great for quickly bringing up an example you're looking for. This repository contains access-controlled folders for every assignment and example code presented in lecture.

- The "**published**" folder will contain example code presented in lecture. This folder can be accessed by anyone in the class under **/published/**.
- You will have your own **personal directory**, accessible only to you, for use in the problem sets.

These are found in the **/users/*your_username*/** directory.

- You will have a **group directory** for each project you work on during the course, found under **/groups/*project_name*/*student-student2-student3*/**.

Setup

To setup your SVN folders and password, visit **https://courses.csail.mit.edu/6.005/svnadmin/**. You must have certificates installed. You only need certificates to access this administration page, however. Any other resource is accessed with your Athena username and your new SVN password. First, set your SVN password on the SVN admin page. Next, create your personal directory by clicking on the appropriate button and pull in the `ps0` assignment. That's it!

SVN in Eclipse

Eclipse has built-in support for working with version control systems, but does not include specific support for SVN. Subclipse, a plug-in for Eclipse, provides this support, allowing you to check out files from a repository, check them in, and use all the other features SVN provides. You can read the Subclipse documentation, also available inside the Eclipse help viewer, to learn more about the features of Subclipse.

On Windows and OS X, there is one extra step you need to take to configure Subclipse. Go to the SVN preferences page ("Window → Preferences... → Team → SVN") and under "SVN interface," choose "SVNKit (Pure Java)." The "JavaHL (JNI)" option will only work if you have a command line Subversion client installed in your `$PATH`.

Connecting to Repositories

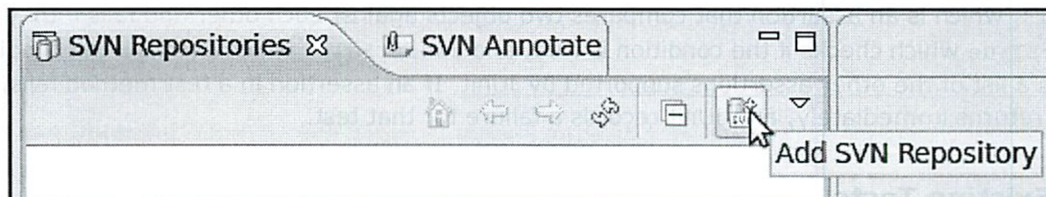
Working with Subclipse begins in the "SVN Repository Exploring" perspective (a perspective in Eclipse is a particular set and arrangement of interface panels). After you install Subclipse and restart Eclipse, switch to this perspective by following their instructions.

Follow the Subclipse documentation for Creating a New Repository Location, with the URL:

The url for the repository is: **https://svn.csail.mit.edu/6.005/fa11**.

Note that to enter the "SVN Repository View" you should click the "Open Perspective" button in the upper right corner of the Eclipse window (it is a small window icon with a plus) and select the "Other..." option. You'll be presented with a dialog listing a number of perspectives, including "SVN Repository Exploring", the view you want.

In order to add the repository, you should click on the Add SVN Repository button shown below.



Checking Out Projects

Checking out a project from the repository once you've added it is straightforward, if that project was originally created in Eclipse. Right now, we'll check out the code for today's problem set. You may be asking: where did this code come from, if it's in my personal directory!? And even if you weren't asking, you should still know that the SVN admin site added the necessary code for you.

Navigate to the `ps0` folder in your directory, right click it, and select "Checkout..." (the documentation on this is under Checking out a Project). You should accept the name of the project unchanged, for this and all problem sets and all group projects.

If Eclipse does not prompt you, switch to the Java perspective with the menu in the top right of the workbench window..

✔**Self-Checkpoint.** You should now be in the Java perspective with the `ps0` project checked out and ready to work on.

Automated Unit Testing with JUnit

JUnit is a widely-adopted Java unit testing library, and we will use it heavily in 6.005. A major component of the 6.005 design philosophy is to decompose problems into minimal, orthogonal units, which can be assembled into the larger modules that form the finished program. One benefit of this approach is that each unit can be tested thoroughly, independently of others, so that faults can be quickly isolated and corrected, as code is rewritten and modules are configured. Unit testing is the technique of writing tests for the smallest testable pieces of functionality, to allow for the flexible and organic evolution of complex, correct systems.

By writing thoughtful unit tests, it is possible to verify the correctness of one's code, and to be confident that the resulting programs behave as expected. In 6.005, we will use JUnit version 4.

The Anatomy of JUnit

JUnit unit tests are written method by method. There is nothing special a class has to do to be used by JUnit; it only need contain methods that JUnit knows to call, which will be referred to as test methods for the remainder of the problem set. Test methods are specified entirely through *annotations*, which may be thought of as keywords (more specifically, they are a type of metadata), that can be attached to individual methods and classes. Though they do not themselves change the meaning of a Java program, at run-time other Java code can detect the annotations of methods and classes, and make decisions accordingly. The Java annotation system, judiciously used, can create dynamic and powerful code. Though we will not deeply explore annotations in 6.005, you will see how other libraries, such as JUnit, make effective use of them.

Look closely at `RulesOf6005Test.java`, and note the `@Test` that precede method definitions. These are examples of annotations. The JUnit library uses these particular annotations to determine which methods to call when running unit tests. The `@Test` annotation denotes a test method; there can be any number in a single class. Even if one test method fails, the others will be run. The test methods contain calls to `assertEquals`, which is an assertion that compares two objects against each other and fails if they are not equal, `assertTrue` which checks if the condition is true, and `assertFalse` which checks if the condition is false. Here is a list of the other assertions supported by JUnit. If an assertion in a test method fails, that test method returns immediately, and JUnit records a failure for that test.

Running Existing Tests

To run the tests in `RulesOf6005Test`, simply right click on the `RulesOf6005Test.java` file in either your Package Explorer, Project View, or Navigator View, and mouse-over the 'Run As' option. Click on the 'JUnit Test' option, and you should see the JUnit view appear, with a green bar indicating that all test methods ran successfully.

To see what a test failure looks like, try running RulesOf6005Test.java before making any changes to RulesOf6005. You should now see a red bar in the JUnit view, and if you click on `testHasFeature`, you will

see a stack trace in the bottom box, which provides a brief explanation of what went wrong. In this case, RulesOf6005 threw a Runtime Exception. Double clicking on lines in the Failure Trace will bring up the code for the test that failed. You should now implement the `hasFeature` method and then rerun the RulesOf6005Test tests. If you implemented the function correctly, you will see that a checkmark next to that test. Passing the public JUnit tests does not necessarily mean that your code is perfect. You should still look over the function specifications carefully and feel free to write your own JUnit tests to verify your code.

For a more thorough introduction, O'Reilly has a JUnit and Eclipse tutorial, with screen-shots to help you get acquainted with using JUnit from within Eclipse. The guide was written for JUnit 3, so the code samples use the older (but still supported) JUnit API.

Checking In Changes to Subversion

You should always remember to commit your changes to the repository at the end of the problem set. In fact, it's not a bad idea to commit after every exercise, as soon as you're happy with the code you've written. When you're working on a team, you'll need to commit your code regularly—until you commit, no one else sees the changes you've made. It is, however, not a good idea to commit code that prevents the program from compiling or that makes it less stable. This is called "breaking the build" as it is a major developer *faux pas*.

You should also remember to commit your projects by the deadline. Where other courses ask you to "hand in" your assignments, in 6.005 you will submit all your assignments by checking them into Subversion.

Before checking in changes, it is always good practice to review what you've done. Right-click on the `ps0` folder that holds your work for this problem set, and select "Team → Synchronize with Repository." Eclipse will prompt you to open yet a new perspective that shows how your local working copy compares with the repository (see Team Synchronizing for more explanation). Grey right-facing arrows indicate files you've modified that need checking in.

Right-click the file you changed and select "Open in Compare Editor" to see a detailed differences, or "diff," view that shows exactly what has been changed. Supposing you're satisfied that this is a change you want to check in, right click the project and select "Commit..." if you're in the Team Synchronizing perspective, or "Team → Commit..." from any other. Read the documentation for the Commit dialog box, where you should:

- Write a message that describes your commit, and
- Make sure the set of files to be checked in is what you want—for example, don't commit Java class files, just source files.

Commit your `hasFeature` implementation right now. If all goes well, your changes should be committed to the repository. If you have any doubt about anything you've done up until now, or don't feel confident that your updated code was committed correctly, ask the LAs or staff. Otherwise—on with Java!

✔**Self-Checkpoint.** You should now have `hasFeature` implemented and committed.

Implement `computeGrade`

Run the JUnit tests and commit your code.

Implement `extendDeadline`

You should read the api for Calendar and GregorianCalendar. Run the JUnit tests and commit your code.

Run the `main()`

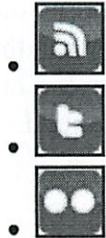
You can run the code in Eclipse by going to Run → Run As... → Java Application, and look for the output in the Console pane at the bottom pane of the screen. The Java Application is going to run the `main` method.

Final commit

This is the end of the problem set. Commit your solutions to your personal Subversion directory, as described above under Checking In Changes to Subversion.

Mike Clark

- Blog
- Photos
- About



JUnit Primer

This article demonstrates a quick and easy way to write and run JUnit test cases and test suites. We'll start by reviewing the key benefits of using JUnit and then write some example tests to demonstrate its simplicity and effectiveness.

Table of Contents

This article contains the following sections:

- Introduction
- Why Use JUnit?
- Design of JUnit
- Step 1: Install JUnit
- Step 2: Write a Test Case
- Step 3: Write a Test Suite
- Step 4: Run the Tests
- Step 5: Organize the Tests
- Testing Idioms
- Training
- Resources

Why Use JUnit?

Before we begin, it's worth asking why we should use JUnit at all. The subject of unit testing always conjures up visions of long nights slaving over a hot keyboard trying to meet the project's test case quota. However, unlike the Draconian style of conventional unit testing, using JUnit actually helps you write code faster while increasing code quality. Once you start using JUnit you'll begin to notice a powerful synergy emerging between coding and testing, ultimately leading to a development style of only writing new code when a test is failing.

what is that and how is JUnit different?

Here are just a few reasons to use JUnit:

- **JUnit tests allow you to write code faster while increasing quality.**

  Yeah, I know, it sounds counter-intuitive, but it's true! When you write tests using JUnit, you'll spend less time debugging, and you'll have confidence that changes to your code actually work. This confidence allows you to get more aggressive about refactoring code and adding new features.

  Without tests, it's easy to become paranoid about refactoring or adding new features because you don't know what might break as a result. With a comprehensive test suite, you can quickly run the tests after changing the code and gain confidence that your changes didn't break anything. If a bug is detected while running tests, the source code is fresh in your mind, so the bug is easily found. Tests written in JUnit help you write code at an extreme pace and spot defects quickly.

- **JUnit is elegantly simple.**

  Writing tests should be simple - that's the point! If writing tests is too complex or takes too much time, there's no incentive to start writing tests in the first place. With JUnit, you can quickly write tests that exercise your code and incrementally add tests as the software grows.

  Once you've written some tests, you want to run them quickly and frequently without disrupting the creative design and development process. With JUnit, running tests is as easy and fast as running a compiler on your code. In fact, you should run your tests every time you run the compiler. The compiler tests the syntax of the code and the tests validate the integrity of the code.

- **JUnit tests check their own results and provide immediate feedback.**

  Testing is no fun if you have to manually compare the expected and actual result of tests, and it slows you down. JUnit tests can be run automatically and they check their own results. When you run tests, you get simple and immediate visual feedback as to whether the tests passed or failed. There's no need to manually comb through a report of test results.

- **JUnit tests can be composed into a hierarchy of test suites.**

  JUnit tests can be organized into test suites containing test cases and even other test suites. The composite behavior of JUnit tests allows you to assemble collections of tests and automatically regression test the entire test suite in one fell swoop. You can also run the tests for any layer within the test suite hierarchy.

- **Writing JUnit tests is inexpensive.**

  Using the JUnit testing framework, you can write tests cheaply and enjoy the convenience offered by the testing framework. Writing a test is as simple as writing a method that exercises the code to be tested and defining the expected result. The framework provides the context for running the test automatically and as part of a collection of other tests. This small investment in testing will continue to pay you back in time and quality.

- **JUnit tests increase the stability of software.**

  The fewer tests you write, the less stable your code becomes. Tests validate the stability of the software and instill confidence that changes haven't caused a ripple-effect through the software. The tests form the glue of the structural integrity of the software.

- **JUnit tests are developer tests.**

  JUnit tests are highly localized tests written to improve a developer's productivity and code quality. Unlike functional tests, which treat the system as a black box and ensure that the software works as a whole, unit tests are written to test the fundamental building blocks of the system from the inside out.

  Developer's write and own the JUnit tests. When a development iteration is complete, the tests are promoted as part and parcel of the delivered product as a way of communicating, "Here's my deliverable and the tests which validate it."

- **JUnit tests are written in Java.**

  Testing Java software using Java tests forms a seamless bond between the test and the code under test. The tests become an extension to the overall software and code can be refactored from the tests into the software under test. The Java compiler helps the testing process by performing static syntax checking of the unit tests and ensuring that the software interface contracts are being obeyed.

- **JUnit is free!**

What are these?

Design of JUnit

JUnit is designed around two key design patterns: the *Command* pattern and the *Composite* pattern.

A `TestCase` is a command object. Any class that contains test methods should subclass the `TestCase` class. A `TestCase` can define any number of public `testXXX()` methods. When you want to check the expected and actual test results, you invoke a variation of the `assert()` method.

`TestCase` subclasses that contain multiple `testXXX()` methods can use the `setUp()` and `tearDown()` methods to initialize and release any common objects under test, referred to as the test fixture. Each test runs in the context of its own fixture, calling `setUp()` before and `tearDown()` after each test method to ensure there can be no side effects among test runs.

`TestCase` instances can be composed into `TestSuite` hierarchies that automatically invoke all the `testXXX()` methods defined in each `TestCase` instance. A `TestSuite` is a composite of other tests, either `TestCase` instances or other `TestSuite` instances. The composite behavior exhibited by the `TestSuite` allows you to assemble test suites of test suites of tests, to an arbitrary depth, and run all the tests automatically and uniformly to yield a single pass or fail status.

Step 1: Install JUnit

1. First, download the latest version of JUnit, referred to below as `junit.zip`.
2. Then install JUnit on your platform of choice:

   Windows

   To install JUnit on Windows, follow these steps:

1. Unzip the `junit.zip` distribution file to a directory referred to as `%JUNIT_HOME%`.
2. Add JUnit to the classpath:

```
set CLASSPATH=%JUNIT_HOME%\junit.jar
```

Unix (bash)

To install JUnit on Unix, follow these steps:

1. Unzip the `junit.zip` distribution file to a directory referred to as `$JUNIT_HOME`.
2. Add JUnit to the classpath:

```
export CLASSPATH=$JUNIT_HOME/junit.jar
```

3. Test the installation by using either the textual or graphical test runner to run the sample tests distributed with JUnit.

Note: The sample tests are not contained in the `junit.jar`, but in the installation directory directly. Therefore, make sure that the JUnit installation directory is in the CLASSPATH.

To use the textual test runner, type:

```
java junit.textui.TestRunner junit.samples.AllTests
```

To use the graphical test runner, type:

Done in Eclipse

```
java junit.swingui.TestRunner junit.samples.AllTests
```

All the tests should pass with an "OK" (textual runner) or a green bar (graphical runner). If the tests don't pass, verify that `junit.jar` is in the CLASSPATH.

Step 2: Write a Test Case

First, we'll write a test case to exercise a single software component. We'll focus on writing tests that exercise the component behavior that has the highest potential for breakage, thereby maximizing our return on testing investment.

To write a test case, follow these steps:

1. Define a subclass of `TestCase`.
2. Override the `setUp()` method to initialize object(s) under test.
3. Optionally override the `tearDown()` method to release object(s) under test.
4. Define one or more public `testXXX()` methods that exercise the object(s) under test and assert expected results.

↑ it does not auto do

The following is an example test case:

```
import junit.framework.TestCase;
```

← they do

```
public class ShoppingCartTest extends TestCase {

    private ShoppingCart cart;
    private Product book1;

    /**
     * Sets up the test fixture.
     *
     * Called before every test case method.
     */
    protected void setUp() {

        cart = new ShoppingCart();

        book1 = new Product("Pragmatic Unit Testing", 29.95);

        cart.addItem(book1);
    }

    /**
     * Tears down the test fixture.
     *
     * Called after every test case method.
     */
    protected void tearDown() {
        // release objects under test here, if necessary
    }

    /**
     * Tests emptying the cart.
     */
    public void testEmpty() {

        cart.empty();

        assertEquals(0, cart.getItemCount());
    }

    /**
     * Tests adding an item to the cart.
     */
    public void testAddItem() {

        Product book2 = new Product("Pragmatic Project Automation", 29.95);
        cart.addItem(book2);

        double expectedBalance = book1.getPrice() + book2.getPrice();

        assertEquals(expectedBalance, cart.getBalance(), 0.0);

        assertEquals(2, cart.getItemCount());
    }

    /**
     * Tests removing an item from the cart.
     *
     * @throws ProductNotFoundException If the product was not in the cart.
     */
    public void testRemoveItem() throws ProductNotFoundException {

        cart.removeItem(book1);
```

```
        assertEquals(0, cart.getItemCount());
    }

    /**
     * Tests removing an unknown item from the cart.
     *
     * This test is successful if the
     * ProductNotFoundException is raised.
     */
    public void testRemoveItemNotInCart() {

        try {

            Product book3 = new Product("Pragmatic Version Control", 29.95);
            cart.removeItem(book3);

            fail("Should raise a ProductNotFoundException");

        } catch(ProductNotFoundException expected) {
            // successful test
        }
    }
}
```

(The complete source code for this example is available in the Resources section).

Step 3: Write a Test Suite

Next, we'll write a test suite that includes several test cases. The test suite will allow us to run all of its test cases in one fell swoop.

To write a test suite, follow these steps:

1. Write a Java class that defines a static `suite()` factory method that creates a `TestSuite` containing all the tests.
2. Optionally define a `main()` method that runs the `TestSuite` in batch mode.

The following is an example test suite:

```
import junit.framework.Test;
import junit.framework.TestSuite;

public class EcommerceTestSuite {

    public static Test suite() {

        TestSuite suite = new TestSuite();

        //
        // The ShoppingCartTest we created above.
        //
        suite.addTestSuite(ShoppingCartTest.class);

        //
        // Another example test suite of tests.
        //
        suite.addTest(CreditCardTestSuite.suite());
```

```
        //
        // Add more tests here
        //

        return suite;
    }

    /**
     * Runs the test suite using the textual runner.
     */
    public static void main(String[] args) {
        junit.textui.TestRunner.run(suite());
    }
}
```

Step 4: Run the Tests

Now that we've written a test suite containing a collection of test cases and other test suites, we can run either the test suite or any of its test cases individually. Running a `TestSuite` will automatically run all of its subordinate `TestCase` instances and `TestSuite` instances. Running a `TestCase` will automatically invoke all of its public `testXXX()` methods.

JUnit provides both a textual and a graphical user interface. Both user interfaces indicate how many tests were run, any errors or failures, and a simple completion status. The simplicity of the user interfaces is the key to running tests quickly. You should be able to run your tests and know the test status with a glance, much like you do with a compiler.

To run our test case using the textual user interface, use:

```
java junit.textui.TestRunner ShoppingCartTest
```

The textual user interface displays "OK" if all the tests passed and failure messages if any of the tests failed.

To run the test case using the graphical user interface, use:

```
java junit.swingui.TestRunner ShoppingCartTest
```

The graphical user interface displays a Swing window with a green progress bar if all the tests passed or a red progress bar if any of the tests failed.

The `EcommerceTestSuite` can be run similarly: Eclipse auto does

```
java junit.swingui.TestRunner EcommerceTestSuite
```

Step 5: Organize the Tests

The last step is to decide where the tests will live within our development environment.

Here's the recommended way to organize tests:

1. Create test cases in the same package as the code under test. For example, the `com.mydotcom.ecommerce` package would contain all the application-level classes as well as the

test cases for those components.

2. To avoid combining application and testing code in your source directories, create a mirrored directory structure aligned with the package structure that contains the test code.

3. For each Java package in your application, define a `TestSuite` class that contains all the tests for validating the code in the package.

4. Define similar `TestSuite` classes that create higher-level and lower-level test suites in the other packages (and sub-packages) of the application.

5. Make sure your build process includes the compilation of all tests. This helps to ensure that your tests are always up-to-date with the latest code and keeps the tests fresh.

By creating a `TestSuite` in each Java package, at various levels of packaging, you can run a `TestSuite` at any level of abstraction. For example, you can define a `com.mydotcom.AllTests` that runs all the tests in the system and a `com.mydotcom.ecommerce.EcommerceTestSuite` that runs only those tests validating the e-commerce components.

The testing hierarchy can extend to an arbitrary depth. Depending on the level of abstraction you're developing at in the system, you can run an appropriate test. Just pick a layer in the system and test it!

Here's an example test hierarchy:

```
AllTests (Top-level Test Suite).
    SmokeTestSuite (Structural Integrity Tests)
        EcommerceTestSuite
            ShoppingCartTestCase
            CreditCardTestSuite
                AuthorizationTestCase
                CaptureTestCase
                VoidTestCase
            UtilityTestSuite
                MoneyTestCase
        DatabaseTestSuite
            ConnectionTestCase
            TransactionTestCase
    LoadTestSuite (Performance and Scalability Tests)
        DatabaseTestSuite
            ConnectionPoolTestCase
        ThreadPoolTestCase
```

Testing Idioms

Keep the following things in mind when writing JUnit tests:

- The software does well those things that the tests check.
- Test a little, code a little, test a little, code a little...
- Make sure all tests always run at 100%.
- Run all the tests in the system at least once per day (or night).

- Write tests for the areas of code with the highest probability of breakage.
- Write tests that have the highest possible return on your testing investment.
- If you find yourself debugging using `System.out.println()`, write a test to automatically check the result instead.
- When a bug is reported, write a test to expose the bug.
- The next time someone asks you for help debugging, help them write a test.
- Write unit tests before writing the code and only write new code when a test is failing.

Resources

- JUnit - The official JUnit website
- JUnit FAQ - Frequently asked questions and answers
- A Dozen Ways to Get the Testing Bug by Mike Clark (java.net, 2004)
- Pragmatic Unit Testing by Andy Hunt and Dave Thomas (The Pragmatic Programmers, 2003)

Copyright © 1999-2011 Clarkware Consulting, Inc. All rights reserved.

6.005 Elements of Software Construction | Fall 2011
Problem Set 1: Pi Poetry
Due: Thursday, September 15 2011, 11:59 PM

write test first

The purpose of this problem set is to give you practice with test-first programming: given specifications, writing unit tests and implementing the code to meet the specification.

The problems are presented in a logical order, but you will receive credit for any parts of the problem set completed, even if you did not complete the prerequisites for that part. So, work ahead if you are stuck on any component of the problem set.

This problem set will have you write tests for methods, then implement those methods. We will be grading both the tests you wrote, as well as the methods themselves. We expect you to write as many tests as necessary to test the methods as we have defined them in the specifications, keeping in mind that many small tests are more useful than a few large tests.

Do not change the signatures or specifications of any methods, classes, or packages that we have provided you. Your code will be tested automatically, and will break our testing suite if you do so.

This problem set is designed so that it can be done with completely stateless functions. All of the functions we have provided you to implement are labeled static. You shouldn't need to introduce class variables or non-static functions in this problem set.

To get started, pull out the problem set code from SVN admin.

Overview

The theme of this project is to find English words in the digits of Pi. To do this, we'll:

1. Compute the fractional digits of Pi
2. Convert the digits of Pi into a numeric base more suitable for word-finding -- i.e., base 26.
3. Transform the digits of Pi into an alphabetic String of letters
4. Find words in that particular encoding of Pi

After the basic version of the code works, we'll work on improving it to get better word coverage -- changing the mapping of digits to letters so that you're more likely to find interesting words.

This problem set is designed to acquaint you with the notion of test-first programming, so your workflow for this problem set should be:

1. Carefully look at the specification of the method you are looking to implement.
2. Write a battery of tests to test the method you are about to write, checking all of the edge conditions for the specification.
3. Write the actual method we specified.

Problem 1: Pi Generation

This part of the problem set will generate an arbitrary number of digits of Pi, so we have data to search through. We will be implementing the Bailey-Borwein-Plouffe formula. Because this algorithm lets us generate arbitrary digits of Pi, it is much easier to test than other Pi generation methods.

Note that the BBP algorithm returns digits of Pi in base-16. Throughout Computer Science, base-16 is commonly referred to as hexadecimal (or hex). Hex is usually expressed using 0-9 and A-F to represent digits, with A = 10, B = 11, ..., F = 15. You'll see that terminology used in the problem set.

We have provided you with almost all of the implementation of the algorithm, the bulk of which can be found in PiGenerator.java. The implementation was ported from the integer-version of the Python code located here. You can read the explanation at that link for the spirit of the implementation we have provided you.

For this part of the problem, you'll have to test and implement computePiInHex() and powerMod() in PiGenerator. Assume that the implementation of piTerm() and piDigit() we've provided you are correct.

not always something that can be assumed!

a. [5 points] Implement tests for powerMod() and place them in PiGeneratorTest. One test has been added for you already.

b. [10 points] Implement powerMod() in PiGenerator.

c. [5 points] Implement tests for computePiInHex() and place them in PiGeneratorTest. You can find some of the hexadecimal expansion of Pi here, if you want to use it as a reference for your test cases.

d. [5 points] Implement computePiInHex() in PiGenerator. Note that this function should only return the fractional digits of Pi, and not the leading 3.

Executing Main.java now should print you some of the hexadecimal digits of Pi. You should verify that the output is what you expect. You can modify PI_PRECISION at the top of the file to change how many digits of Pi to generate; generating less digits will be faster.

Commit to Subversion. Once you're happy with your solution to this problem, commit your code! Committing frequently -- whenever you've fixed a bug or added a working and tested feature -- is a good way to use version control, and will be a good habit to have for your team projects.

Problem 2: Transforming Pi

In order to find words in the digits of Pi, we'll first need to convert the hex digits of Pi into something more useful. As a first try, we'll convert the hex digits into base-26, then do the straightforward mapping of numbers to letters. This part of the problem set will implement BaseTranslator.convertBase()

a. [10 points] A starting test has already been written for you in BaseTranslatorTest.java. Add additional tests to BaseTranslatorTest that tests the functionality of convertBase(). Ensure that all boundary conditions are tested, and that the behavior described in the specifications for those functions are complied with.

null, non strings, what else?

b. [15 points] Implement convertBase(), verifying that the tests you've written for it pass.

Executing Main.java should print you the base-26 digits of Pi. Verify that the numbers are what you expect.

Problem 3: Converting Pi to Characters

Now that we have Pi in base-26, it is a straightforward task to convert it to a string of letters. This part of the problem set will implement DigitsToStringConverter.convertDigitsToStrings().

a. [5 points] A starting test has already been written for you in DigitsToStringConverterTest.java. Add additional tests to DigitsToStringConverterTest that tests the functionality of convertDigitsToString().

b. [10 points] Implement convertDigitsToString()

Executing Main.java should print you the translation of base-26 Pi into a-z characters.

Problem 4: Finding Words

Now that we have an alphanumeric string, we want to find words in it.

a. [5 points] A starting test has already been written for you in WordFinderTest.java. Add additional tests to WordFinderTest that tests the functionality of getSubstrings(). Ensure that all boundary conditions are tested, and that the behavior described in the specifications for those functions are complied with.

b. [5 points] Implement getSubstrings(), verifying that it conforms to the listed specification, and that all of your tests pass.

Executing Main.java now should show you a list of the words that were found in the digits of Pi! It should also tell you what percentage of words were found from the word list included in the assignment.

Problem 5: Alphabet Generation Revisited

As you can see, we don't do very well with finding most words in the digits of Pi. Part of the reason is that the alphabet we're using is not very smart; "z"s occur with roughly the same frequency as "e"s. This problem will explore one way to improve our implementation.

We'll use the word list that is included in the code to take a guess at how often each character occurs relative to the other characters, and we'll weigh the output alphabet we're translating with in favor of more frequently occurring characters.

a. [10 points] Implement tests for generateFrequencyAlphabet() in AlphabetGeneratorTest.

b. [15 points] Implement generateFrequencyAlphabet().

Executing Main.java now should show a list of words that were found in the digits of Pi using the alternative alphabet. The coverage you should get for this implementation should be higher than in the basic one.

Before You're Done...

Double check that you didn't change the signatures of any of the code we gave you.

Make sure the code you implemented doesn't print anything to System.out. It's a helpful debugging feature, but writing output is a definite side effect of methods, and none of the methods we gave you to write should produce any side effects.

Make sure you don't have any outdated comments in the code you turn in. In particular, get rid of blocks of code that you may have commented out when doing the pset, and get rid of any TODO comments that are no longer TODOs.

Make sure your code compiles, and all the methods you've implemented pass all the tests that you added.

Does your code compile without warnings? In particular, you should have no unused variables, and no unneeded imports.

Make sure you check the last version of your code in SVN is the version you want to be graded.

6.005 Doing P-Set 1 9/11

need to put @ in front of test cases
can have multiple test sections
or put multiple tests inside

What to test?

- negitive
0
nulls
All non ints ← Java auto checks?

Go slow + really learn
So apparently test is mon lecture
-so just do what I have so far
Took me 50 min to top part - but leaning about test

②

Now pi in Hex()

~~Am~~ Simpler to test

Implement BBP formula

$$\pi = \sum_{k=0}^{\infty} \left[\frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \right]$$

So is this precision?

What is result?

Int - can't be if in Hex

I stuck w/ hex in Java

Should have provided a test for us

I'm not quite sure what they want!

Ok array of ints, one per item in array

0 - 16

return all values

So no Σ - we do that later

(3)

Go to lab?

I think getting pointer errors...

How do you just run functions?

I think its becaus 16^k goes to ∞ fast

How to fix?

Oh power mod is used

Ohh did they write the BBP function already?

Then we just need to convert in hex?

But they do $.16$ - not $\frac{1}{16^k}$ - weird

Oh further down - they do something weird using power mod

Now how do you start? $k = 0$ or 1?
3 2 ↑ start here

Now we are in typical (6.01) array issues

Now good to 8th character

Oh did not have long test cases for exp

And w/ new method need to build if b==0

Now good up to 100

~~Also 2~~

Need to convert hex to regular
- how to do that

Don't think wee need to

Ok all good part 1 6.10

(Should have committed half stuff
- oh well)

BaseTranslator.java

```
1 package piwords;
2
3 public class BaseTranslator {
4     /**
5      * Converts an array where the ith digit corresponds to (1 / baseA)^(i
  + 1)
6      * digits[i], return an array output of size precisionB where the ith
  digit
7      * corresponds to (1 / baseB)^(i + 1) * output[i].
8      *
9      * Stated in another way, digits is the fractional part of a number
10     * expressed in baseA with the most significant digit first. The
  output is
11     * the same number expressed in baseB with the most significant digit
  first.
12     *
13     * To implement, logically, you're repeatedly multiplying the number
  by
14     * baseB and chopping off the most significant digit at each
  iteration:
15     *
16     * for (i < precisionB) {
17     *   1. Keep a carry, initialize to 0.
18     *   2. From RIGHT to LEFT
19     *      a. x = multiply the ith digit by baseB and add the carry
20     *      b. the new ith digit is x % baseA
21     *      c. carry = x / baseA
22     *   3. output[i] = carry
23     *
24     * If digits[i] < 0 or digits[i] >= baseA for any i, return null
25     * If baseA < 2, baseB < 2, or precisionB < 1, return null
26     *
27     * @param digits The input array to translate. This array is not
  mutated.
28     * @param baseA The base that the input array is expressed in.
29     * @param baseB The base to translate into.
30     * @param precisionB The number of digits of precision the output
  should
31     *                   have.
32     * @return An array of size precisionB expressing digits in baseB.
33     */
34    public static int[] convertBase(int[] digits, int baseA,
35                                    int baseB, int precisionB) {
36        // TODO: Implement (Problem 2.b)
```

```
37          return null;
38      }
39  }
40
```

So test

.1 in base 2

is .25 in base 10

following

$0 \times \left(\frac{1}{2}\right)^1 + 1 \cdot \left(\frac{1}{2}\right)^2 = .25$

i+1

1 base A

inputs/digits in reverse!

baseA = 2

baseB = 10

precision = 2

is subdivided .1 → {1, 0}

but where is fraction

- no initial # already divided like that

where does base B matter?

(5)

Part 2

Here is mapping
- weird

∴ Very confusing what you should actually do even when looking at the mapping

Spending a lot of my time on edge cases / testing

What is a fractional part of the number

Run through

in = 0, 1   2, 10, 2
A, B, P

Output = [_, _]

Carry = 0

for 0, 1, 2

$x = 1 \cdot 10 + 0$

digit [0] = 10 mod 2 = 0

carry = 10/2 = 5

output [0] = 5

(6)

for → i = 1

x = 0 · 10 + 5

digit [1] = 5 mod 2 = 1

carry = 5/2 = 2

output [1] = ~~0~~ 2

~~for i = 2~~

~~x = null + 2~~

(, reverse — supposed to be 2,5

Code got [0]

Oh did not see from right to left

Now it works for my mesily test case

(7)

Make more test cases

Input .50 base 2 to 10, pre 2

$\frac{1}{2}^{(0+1)} \cdot 0 + \left(\frac{1}{2}\right)^{(1+1)} \cdot 5 =$

↑ who made that complicated formula up in that it returns just the digit

$= \underline{4}.20448$

Input .55 2 to 10, pre 2

$\frac{1}{2} \cdot 5 + \frac{1}{2}^{2} \cdot 5 =$

Oh 5 > 2 does not work

So .02 base 2 - if = 2 nope

.02 base 3 to 10, pre 2

$\left(\frac{1}{3}\right)^{2} \cdot 2 = .2222$

It gets 26

(18)

try again

$i = 1$

$x = digit[1] \cdot base B + carry$

$= 2 \cdot 10 + 0$

$= 20$

$digit[1] = 20 \% 2 = 0$

$carry = 20 / 2 = 10$

output [1] = ~~carry~~ 10

?? should not be!

That digits thing is wrong!

$i = 0$

$x = 0 \cdot 10 + 10$

$= 10$

$digit[0] = 10 \% 2 = 0$

$carry = 10 / 2 = 5$

$output[0] = 5$

5, 10

↑ but I get 2, 6.

(9)

Pizzza has inner carry loops

Yeah iterate twice once for digits

Once for precision

(It would really be nice to know upfront)

Now my other by hand one matches

Now let me try precision 3

Part 3 Convert Pi to ~~Strings~~ Chars

"straight forward"

I'm liking to do tests 2nd

but I see why 1st

Even though they write everything to test

But why no junit tests found error

It was pretty simple

I don't get why test does not work!

Now it magically works!

10a

DigitsToStringConverter.java

```
1 package piwords;
2 
3 public class DigitsToStringConverter {
4     /**
5      * Given a list of digits, a base, and an mapping of digits of that
 base to
6      * chars, convert the list of digits into a character string by
 applying the
7      * mapping to each digit in the input.
8      *
9      * If digits[i] >= base or digits[i] < 0 for any i, consider the input
10     * invalid, and return null.
11     * If alphabet.length != base, consider the input invalid, and return
 null.
12     *
13     * @param digits A list of digits to encode. This object is not
 mutated.
14     * @param base The base the digits are encoded in.
15     * @param alphabet The mapping of digits to chars. This object is not
16     *                 mutated.
17     * @return A String encoding the input digits with alphabet.
18     */
19    public static String convertDigitsToString(int[] digits, int base,
20                                               char[] alphabet) {
21        // TODO: Implement (Problem 3.b)
22        return "";
23    }
24 }
25 
```

(11)

Part 4 Finding words

The errors are actually surprisingly useful

(1b)

WordFinder.java

```
1 package piwords;
2
3 import java.util.HashMap;
5
6 public class WordFinder {
7     /**
8      * Given a String (the haystack) and an array of Strings (the needles),
9      * return a Map<String, Integer>, where keys in the map correspond to
10     * elements of needles that were found as substrings of haystack, and the
11     * value for each key is the lowest index of haystack at which that needle
12     * was found. A needle that was not found in the haystack should not be
13     * returned in the output map.
14     *
15     * @param haystack The string to search into.
16     * @param needles The array of strings to search for. This array is not
17     *                mutated.
18     * @return The list of needles that were found in the haystack.
19     */
20    public static Map<String, Integer> getSubstrings(String haystack,
21                                                     String[] needles) {
22        // TODO: Implement (Problem 4.b)
23        return new HashMap<String, Integer>();
24    }
25 }
26
```

Need to learn Map format

So for each Needle
Search for it step through chars
if found, add it

That is nieve method - too bad

well substring

(12)

Part 5 Alphabet generation Revisited

(Took me a while - but did not write much)

The letters thing which took me a while to write turned out ~~not to do anything!~~ not to be needed!

On Tortise SVN communicates w/ Eclipse seamless

```java
package piwords;

public class AlphabetGenerator {
    /**
     * Given a numeric base, return a char[] that maps every digit that is
     * representable in that base to a lower-case char.
     *
     * This method will try to weight each character of the alphabet
     * proportional to their occurrence in words in a training set.
     *
     * This method should do the following to generate an alphabet:
     *   1. Count the occurrence of each character a-z in trainingData.
     *   2. Compute the probability of each character a-z by taking
     *      (occurrence / total_num_characters).
     *   3. The output generated in step (2) is a PDF of the characters in the
     *      training set. Convert this PDF into a CDF for each character.
     *   4. Multiply the CDF value of each character by the base we are
     *      converting into.
     *   5. For each index 0 <= i < base,
     *      output[i] = (the first character whose CDF * base is > i)
     *
     * A concrete example:
     *   0. Input = {"aaaaa..." (302 "a"s), "bbbbb..." (500 "b"s),
     *               "ccccc..." (198 "c"s)}, base = 93
     *   1. Count(a) = 302, Count(b) = 500, Count(c) = 193
     *   2. Pr(a) = 302 / 1000 = .302, Pr(b) = 500 / 1000 = .5,
     *      Pr(c) = 198 / 1000 = .198
     *   3. CDF(a) = .302, CDF(b) = .802, CDF(c) = 1
     *   4. CDF(a) * base = 28.086, CDF(b) * base = 74.586, CDF(c) * base = 93
     *   5. Output = {"a", "a", ... (29 As, indexes 0-28),
     *                "b", "b", ... (46 Bs, indexes 29-74),
     *                "c", "c", ... (18 Cs, indexes 75-92)}
     *
     * The letters should occur in lexicographically ascending order in the
     * returned array.
     *   - {"a", "b", "c", "c", "d"} is a valid output.
     *   - {"b", "c", "c", "d", "a"} is not.
     *
     * If base >= 0, the returned array should have length equal to the size of
     * the base.
     *
     * If base < 0, return null.
     *
     * If a String of trainingData has any characters outside the range a-z,
     * ignore those characters and continue.
     *
     * @param base A numeric base to get an alphabet for.
     * @param trainingData The training data from which to generate frequency
     *                     counts. This array is not mutated.
     * @return A char[] that maps every digit of the base to a char that the
     *         digit should be translated into.
     */
```

```java
    public static char[] generateFrequencyAlphabet(int base,
                                                    String[] trainingData) {
        // TODO: Implement (Problem 5.b)
        return null;
    }
}
```

9/12

Code Reviewing

In this course, you will read your classmates' code and give them comments about it. This document describes the whys and hows of the 6.005 code reviewing process.

You can't learn how to write without learning how to read -- and programming is no exception. Code reviewing is widely used in software development, both in industry and in open source projects. Some companies like Google have instituted review-before-commit as a required policy. You can't get a line of code into the Google source code repository unless another Google engineer has read it, given feedback about it, and signed off on it.

The benefits of code review in practice are several. Reviewing helps find bugs, in a way that's complementary to other techniques (like static checking, testing, assertions, and reasoning). Reviewing uncovers code that is confusing, poorly documented, unsafe, or otherwise not ready for maintenance or future change. Reviewing also spreads knowledge through an organization, allowing developers to learn from each other by explicit feedback and by example. So code reviewing is not only a practically important skill that you will need in the real world, but also a learning opportunity.

Code reviewing is also one way to broaden participation in the course. We will be inviting alums of 6.005 (and its predecessor 6.170) to participate in reviewing code. Many of them are out working in industry or startups, and they have experience in the trenches of software development that they can share.

What to look for

Although you can make comments about anything you think is relevant, the primary goal of this class is to learn how to write code that is safe from bugs, easy to understand, and ready for change. Read the code with those principles in mind. Here are some concrete examples of problems to look for:

Bugs or potential bugs. Disagreement between code and specification. Off-by-one errors. Risky variable scopes. Optimistic, undefensive programming. ...

Unclear, messy code. Bad variable or method names. Inconsistent indentation. Convoluted control flow (if and while statements) that could be simplified. Packing too much into one line of code, or too much into one method. Failing to comment obscure code. Having too many trivial comments that are simply redundant with the code. ...

Misunderstandings of Java. Misuse of == or .equals(). Misuse of arrays or Lists. ...

Misusing (or failing to use) essential design concepts. Incomplete or incorrect specification for a method or class. Representation exposure for a data abstraction. Immutable datatypes that expose themselves to change. Invariants that aren't really invariant, or aren't even stated. Failure to implement the Object contract correctly (equals and hashCode). ...

Positive comments are also a good thing. Don't be afraid to make comments about things you really like, for example:

Unusually elegant code.

Creative solutions.

Great design.

Process

We will be using a new web-based code-reviewing system called Caesar, built here at MIT. Caesar is different from other code review tools (like Rietveld and Review Board) in several ways. It's designed for reviewing whole programs, rather than changesets. It divides up a program into small pieces (methods), assigns them to multiple reviewers, and supports discussion-based reviewing with threaded comments and up and down votes. It automatically ignores widely-repeated code, like boilerplate or library code, in order to focus attention on code that's worth the reviewers' time. With this system, you will read small bits of code written by several other students, and many other students will each read a small bit of yours.

Here's how the process will typically go. Only problem set code will be reviewed. Team projects will not go through this process.

Thursday midnight: problem set is due.

Friday morning: submitted code is loaded into Caesar, and an announcement goes out by email to all reviewers that the reviewing process is open.

Friday/Saturday: you should visit Caesar, read the methods that you were assigned, and make comments about them. For each method you review, you can:

- make a comment
  by clicking on a line of code or selecting a range of lines, and typing your comment.
- reply to another comment
  by clicking on Reply. You can use this to clarify, expand on, or disagree with comments made by other reviewers.
- upvote a comment
  by clicking the Thumbs-up icon. Use this to agree with a comment made by another reviewer. Some comments in the system are

made automatically by a style checker (Checkstyle), and it's particularly useful for human reviewers to upvote or downvote its comments as appropriate.

- downvote a comment

  by clicking on the Thumbs-down icon. If you're disagreeing with a human, then it's polite and helpful to also Reply to explain why.

Saturday midnight: code reviews are due.

Sunday: course staff (TAs, LAs, lecturer) review the reviews: upvoting, downvoting, replying, and commenting as appropriate, and looking for examples worth highlighting for the whole class.

Monday: Look at your own code! Also, highlights of the code review -- great code, problem code, great review comments -- are collected and shared with everybody through the Piazza forum and the Caesar dashboard.

Privacy & visibility

As a code author, you are anonymous. The system does not display your name with any of the code you wrote. Please don't put your full name, username, email address, or other identifying information in your source code. We try to strip out your name and username if it appears, but this process isn't perfect, especially since some of you have names that are pretty common in source code. Yes, I'm looking at you, little Bobby Tables.

As a reviewer, you are fully identified. The system displays your name and username with every review comment, reply, upvote, or downvote that you were responsible for. All registered users can see your name and what you wrote.

All reviewers, all code, and all comments are visible to registered users. The "all users" link shows a list of all users, and each user has a profile page showing all the comments that they have made. A URL for a chunk of code or a review comment can be viewed by anybody logged in to Caesar, regardless of whether they were assigned as a reviewer for that particular chunk.

Caesar is accessible only to registered users. The system is not open to the net, nor is it indexed by Google. It is only visible to members or alumni of 6.005/6.170.

Reputation

As a reviewer in Caesar, you have a reputation score, shown in parentheses after your name. You earn reputation points as follows:

- +1 pt for each upvote (by another user) of a comment you wrote
- +10-50 pts for writing a comment that is selected for highlighting
- +100 pts for passing 6.005/6.170.

If you start reviewing early, then you are more likely to get upvotes. If you write good comments, then you are more likely to get upvotes.

As a student in 6.005, your numeric reputation score has no effect whatsoever on your grade. Your participation grade does depend heavily on code reviewing, but the reputation score itself is irrelevant to that. Instead, we will base your participation grade on the quantity and quality of the actual reviewing that you've done, by looking at the activity history on your profile page.

Respect

Be polite. Sarcasm, insults, and belittling words have no place in a code review. It doesn't matter whether you're talking about a person (a fellow reviewer or a code author) or about code. Don't call code "stupid," because that transfers all too easily to the author of the code, whether you meant it that way or not.

Be constructive. Don't just criticize, but be helpful: point the way toward solutions. "Hopeless mess" is not a constructive comment; "name the variables more descriptively, e.g. tmp1 is not a great name" is much more constructive.

As a reviewer, you should downvote comments by others that are unconstructive or rude, and serious problems should be referred to the teaching staff.

As a code author, you should read your feedback carefully, and keep an open mind. Don't get defensive. If your reviewers -- who are MIT students and alums -- find your code confusing, then you should consider what this says about its clarity and maintainability in the real world. If you disagree with a comment, you can reply to it and engage the reviewer in a discussion, but keep in mind that this makes your identity visible.

FAQ for reviewers

Does reviewing affect my grade? Yes, it contributes to your participation grade; see the Reputation section above.

Does my reviewing affect other people's grades? Not directly. Problem set grades do not take code reviewing into account, and other reviewers' participation grades are not affected by your upvotes or downvotes. But your reviewing will hopefully help other students become better programmers and acquire a better understanding of the course, and indirectly improve their grades.

How many methods do I need to review, and how much do I need to do on each one? The number you are assigned will depend on the size of the problem set, but will not exceed 10 methods for students (5 for alums). At a minimum, you must do at least one thing on each method assigned to you -- make a comment, reply to a comment, upvote, or downvote.

I don't feel like I know anything. How can I possibly review other people's code? You know more than you think you do. You can read for clarity, comment on places where the code is confusing, look for problems we talked about in lecture or recitation, etc.

What if I can't find anything wrong? You can write a positive comment about something good. Or you can simply say "looks good to me", also known as #lgtm.

What are these "checkstyle" comments? They are generated automatically by a style checker (Checkstyle). Some of these comments might be wrong or inappropriate for the situation, so please use upvoting and downvoting to review Checkstyle too.

How much of the whole program do I have to try to understand? Caesar presents each method in isolation, and asks for comments just for that method. We don't expect you to look at anything else, or to spend time struggling to figure out whether it's correct. If you want to see the rest of the program, you can do so using the "view all code" link. But again, it's not expected.

How can I find out who wrote this code? Code authors are anonymous until they reveal themselves -- e.g. by replying to a comment and stating that they're the code author. You can write a comment asking a code author to reveal their identity, or contact you, if you wish. If there is a serious situation requiring the identity of the code author -- e.g., a potential case of plagiarism -- then bring it to the attention of the teaching staff.

FAQ for code authors

Can somebody use my code to cheat in the class? Since code reviewing doesn't start until after the deadline, this would be a problem only for students with extensions. Those students may not review code until after their extended deadline.

Is it OK to break my anonymity? Yes. Please let the reviewing process end before you get involved in any discussions of your code, but after that, if you want to join the discussion or directly contact one of your reviewers, you can do so.

Can I look at the reviews of my code while reviewing is still in progress? Yes, you can watch your reviews during the reviewing period, but don't jump in defensively. See Tone & Attitude above.

9/12

Missed due to MITCET meeting

6.005 Software Construction
Fall 2011
Prof. Rob Miller

L2: Test-First Programming

Today

Reviewed afterward

- Testing
- Choosing test cases
- Blackbox vs. whitebox testing
- Coverage
- Stubs, drivers, oracles
- Regression testing

Real Programmers Don't Test (?)

Here are the top-5 reasons why Louis Reasoner doesn't want to test his code:

5) I want to get this done fast – testing is going to slow me down.

4) I started programming when I was 2. Don't insult me by testing my perfect code!

3) Testing is for incompetent programmers who cannot hack.

2) We're not Harvard students – our code actually works!

1) "Most of the functions in Graph.java, as implemented, are one or two line functions that rely solely upon functions in HashMap or HashSet. I am assuming that these functions work perfectly, and thus there is really no need to test them." – an excerpt from a 6.170 student's e-mail

haha Something I would write

The biggest problem for Louis is optimism. Change that perspective – look for things that can go wrong, rather than assuming Pollyannishly that all will go right.

Testing isn't rocket science. Except when it is: a famous case was the Ariane 5 launch vehicle, designed and built for the European Space Agency in the 1990s. It self-destructed 37 seconds after its first launch. The reason was a control software bug that went undetected. Ariane 5's guidance software was reused from the Ariane 4, which was a slower rocket. As a result, when the velocity calculation converted from a 64-bit floating point number (a double in Java terminology, though this software wasn't written in Java) to a 16-bit signed integer (a short), it overflowed the small integer and caused an exception to be thrown. The exception handler had been disabled for efficiency reasons, so the guidance software crashed... and without guidance, the rocket did too. The cost of the failure was $1 billion... a cost that might have been prevented with better testing.

In PS1, we have a sequence of steps feeding each other; a stack of methods building on each other. The optimistic hacker's approach to PS1 might build the whole thing before testing any part of it. Contrast with the engineer's approach: isolate each component, build it and test it soundly in isolation, then plug together.

Why Testing is Hard

we want to

- know when product is stable enough to launch

- deliver product with known failure rate (preferably low)
- offer warranty?

but

- it's very hard to measure or ensure quality in software
- residual defect rate after shipping:
  - 1 - 10 defects/kloc (typical)
  - 0.1 - 1 defects/kloc (high quality: Java libraries?)
  - 0.01 - 0.1 defects/kloc (very best: Praxis, NASA)

  example: 1Mloc with 1 defect/kloc means you missed 1000 bugs!

exhaustive testing is infeasible

- space is generally too big to cover exhaustively
- imagine exhaustively testing a 32-bit floating-point multiply operation, a*b
  - there are 2^64 test cases!

statistical testing doesn't work for software

- other engineering disciplines can test small random samples (e.g. 1% of hard drives manufactured) and infer defect rate for whole lot
- many tricks to speed up time (e.g. opening a refrigerator 1000 times in 24 hours instead of 10 years)
- gives known failure rates (e.g. mean lifetime of a hard drive)
- but assumes continuity or uniformity across the space of defects, which is true for physical artifacts
- **this is not true** for software
  - overflow bugs (like Ariane 5) happen abruptly
  - Pentium division bug affected approximately 1 in 9 billion divisions

often confused, but very different

(a) problem of finding bugs in defective code

(b) problem of showing **absence** of bugs in good code

approaches

- testing: good for (a), occasionally (b)
- reasoning: good for (a), also (b)

theory and practice

- for both, you need grasp of basic theory
- good engineering judgment essential too

what are we trying to do?

- find bugs as cheaply and quickly as possible

reality vs. ideal

- ideally, choose one test case that exposes a bug and run it
- in practice, have to run many test cases that "fail" (because they don't expose any bugs)

in practice, conflicting desiderata

- increase chance of finding bug
- decrease cost of test suite (cost to generate, cost to run)

design testing strategy carefully

- know what it's good for (finding egregious bugs) and not good for (security)
- complement with other methods: code review, reasoning, static analysis
- exploit automation (e.g. JUnit) to increase coverage and frequency of testing

? can it auto write test cases?

➢ so behavior must be the "same" for whole sets of inputs

ideal test suite

➢ identify sets of inputs with the same behavior

➢ try one input from each set

multiply : BigInteger × BigInteger → BigInteger

➢ partition BigInteger into:

BigNeg, SmallNeg, -1, 0, 1, SmallPos, BigPos

➢ pick a value from each class

-265, -9 -1, 0, 1, 9, 265

➢ test the 7 × 7 = 49 combinations

max : int × int → int

➢ partition into:

a < b, a = b, a > b

➢ pick value from each class

(1, 2), (1, 1), (2, 1)

intersect : Set × Set → Set

➢ partition Set into:

∅, singleton, many

➢ partition whole input space into:

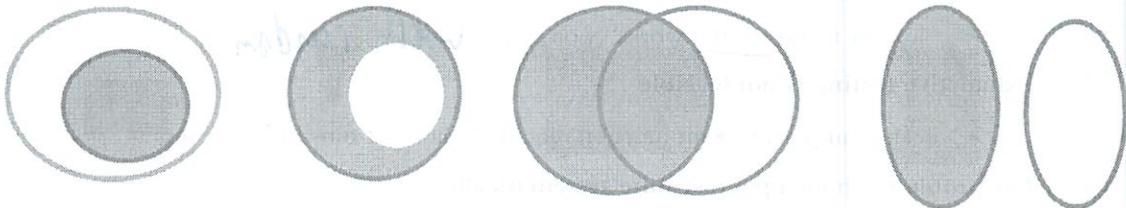
this = that, this ⊆ that, this ⊇ that, this ∩ that ≠ ∅, this ∩ that = ∅

➢ pick values that cover both partitions

{},{} {},{2} {},{2,3,4}

{5},{} {5},{2} {4},{2,3,4}

{2,3},{} {2,3},{2} {1,2},{2,3}



Key idea #2: Boundary testing

➢ include classes at **boundaries** of the input space

- zero, min/max values, empty set, empty string, null

➢ why? because bugs often occur at boundaries

- do it early and often: **test-first programming**

Basic Notions

what's being tested?

- unit testing: individual module (method, class, interface)
- subsystem testing: entire subsystems
- integration, system, acceptance testing: whole system

how are inputs chosen?

- random: surprisingly effective (in defects found per test case), but not much use when most inputs are invalid (e.g. URLs)
- systematic: partitioning large input space into a few representatives
- arbitrary: *not* a good idea, and not the same as random!

how are outputs checked?

- automatic checking is preferable, but sometimes hard (how to check the display of a graphical user interface?)

how good is the test suite?

- coverage: how much of the specification or code is exercised by tests?

when is testing done?

- test-first programming: tests are written first, before the code
- regression testing: a new test is added for every discovered bug, and tests are run after every change to the code

essential characteristics of tests

- modularity: no dependence of test driver on internals of unit being tested
- automation: must be able to run (and check results) without manual effort

Choosing Test Cases

arbitrary testing is not convincing

- "just try it and see if it works" won't fly

exhaustive testing is not feasible

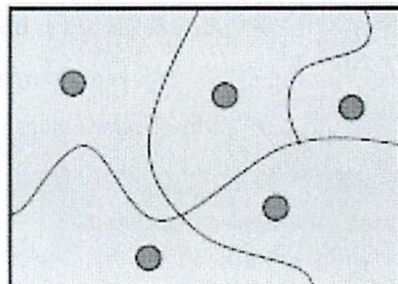
- even a simple int → int function requires billions of runs to test all inputs

key problem: choosing a test suite systematically

- small enough to run quickly
- large enough to validate the program convincingly

Key Idea #1: Partition the Input Space

input space is very large, but program is small



- off-by-one errors
- forget to handle empty container
- overflow errors in arithmetic

basic

Blackbox vs. whitebox

black box testing

- ➢ choosing test data only from spec, without looking at implementation

glass box (white box) testing

- ➢ choosing test data with knowledge of implementation
    - e.g. if implementation does caching, then should test repeated inputs
    - if implementation selects different algorithms depending on the input, should choose inputs that exercise all the algorithms
- ➢ must take care that tests don't *depend* on implementation details
    - e.g. if spec says "throws exception if the input is poorly formatted", your test shouldn't check specifically for a NullPtrException just because that's what the current implementation does
- ➢ good tests should be **modular** -- depending only on the spec, not on the implementation

    - ➢ **max : List<int> → int**

well if the function is

```
/**
 * Find the largest element in a list.
 * @param l list of elements.  Requires l to be nonempty
 * and all elements to be nonnegative.
 * @return the largest element in l
 */
public static int max(List<Integer> l) { ... }
```

- list length: ~~length 0~~, length 1, length 2+
- max position: start, middle, end of list
- value of max: ~~MIN_INT~~, ~~negative~~, 0, positive, MAX_INT

r all of these

diff sizes

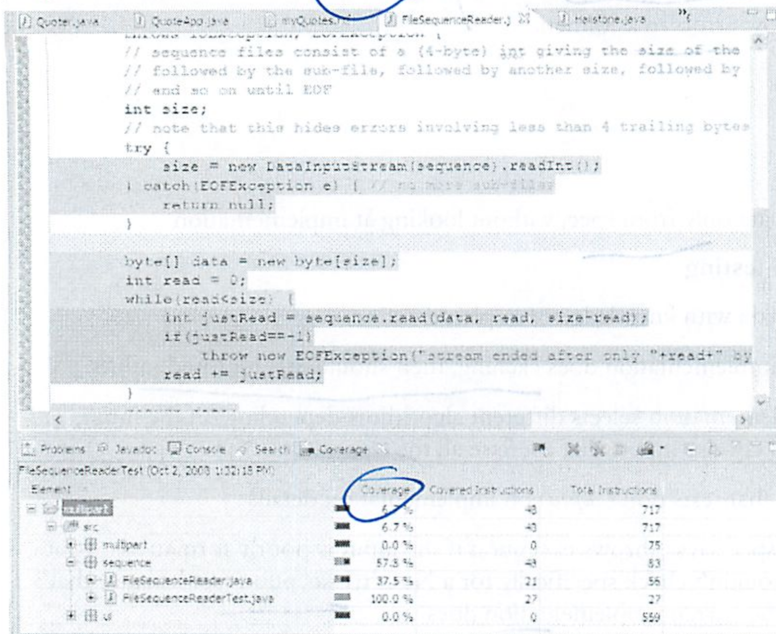
Coverage

Three kinds of coverage:

- all-statements: is every statement run by some test case?
- all-branches: if every direction of an if or while statement (true or false) taken by some test case?
- all-paths: is every possible combination of branches – every path through the program – taken by some test case?

specification coverage vs. code coverage

a typical code coverage tool (EclEmma plugin for Eclipse):



industry practice

- all-statements is common goal, rarely achieved (due to unreachable code)
- all branches if possible: safety critical industry has more arduous criteria (eg, "MCDC", modified decision/condition coverage)
- all-paths is infeasible

using coverage for white-box testing

- generate black-box test cases until code coverage is sufficient

Test frameworks

driver

- just runs the tests
- must design unit to be drivable!
- eg: program with GUI should have API

stub

- replaces other system components
- allows reproducible behaviours (esp. failures)

oracle

- determines if result meets spec
- preferably automatic and fast

- varieties: computable predicate (e.g. is the result odd?), comparison with literal (e.g. must be 5), manual examination (by a human)
- in regression testing, can use previous results as "gold standard"

Test-first programming

write tests before coding

- specifically, for every method or class:

  1) write specification

  2) write test cases that cover the spec

  3) implement the method or class

  4) once the tests pass (and code coverage is sufficient), you're done

writing tests first is a good way to understand the spec

- think about partitioning and boundary cases
- if the spec is confusing, write more tests
- spec can be buggy too
  - incorrect, incomplete, ambiguous, missing corner cases
  - trying to write tests can uncover these problems

and fill in spec

Regression Testing

whenever you find and fix a bug

- store the input that elicited the bug
- store the correct output
- add it to your test suite

good idea

why regression tests help

- helps to populate test suite with good test cases
  - remember that a test is good if it elicits a bug – and every regression test did in one version of your code
- protects against reversions that reintroduce bug
- the bug may be an easy error to make (since it happened once already)

test-first debugging

- when a bug arises, immediately write a test case for it that elicits it
- once you find and fix the bug, the test case will pass, and you'll be done

How to avoid debugging

first defense against bugs is to make them impossible

- Java makes buffer overflow bugs impossible
- static typing eliminates many runtime type errors
- immutable objects like Strings and URLs can be passed around and shared without fear that they will be modified
- immutable (final) references guarantee that you won't

if we can't prevent bugs, we can try to localize them to a small part of the program

- fail fast: the earlier a problem is observed, the easier it is to fix
- assertions: catch bugs early, before failure has a chance to contaminate (and be obscured by) further computation

  in Java: assert *boolean-expression*

  note that you must enable assertions with -ea
- unit testing: when you test a module in isolation, you can be confident that any bug you find is in that unit (or in the test driver)
- regression testing: run tests as often as possible when changing code.

  if a test fails, the bug is probably in the code you just changed

when localized to a single method or small module, bugs may be found simply by studying the program text

Code review

other eyes looking at the code can find bugs

code review

careful, systematic study of source code by others (not original author)

analogous to proofreading an English paper

look for bugs, poor style, design problems, etc.

formal inspection: several people read code separately, then meet to discuss it

lightweight methods: over-the-shoulder walkthrough, or by email

many dev groups require a code review before commit

code review complements other techniques

code reviews can find many bugs cheaply

also test the understandability and maintainability of the code

three proven techniques for reducing bugs: static checking, code reviews, testing

An example of test-first programming

First, the spec:

```
/**
 * Find the first occurrence of x in sorted array a.
 * @param x value to find
 * @param a array sorted in increasing order (a[0] <= a[1] <= ... <= a[n-
1])
 * @return lowest i such that a[i]==x, or -1 if x not found in a.
 */
public static int find(int x, int[] a) {...}
```

Then, the test cases:

// Testing strategy for i = search(x, a):
// partition the space of (x, a, i) as follows

// x: neg, 0, pos ← yeah (-), 0, (+)

// a.length: 0, 1, 2+

// a.vals: neg, 0, pos; all same, increasing;

// i: 0, middle, n-1, -1 Testing here meaning

- // x=2, a=[-1, 1, 3], i=-1
- // x=0, a=[0], i=0
- // x=1, a=[-1, 1, 3], i=1
- // x=-1, a=[-1, 1, 3], i=0
- // x=3, a=[-1, 1, 3], i=n-1
- // x=2, a=[-1, 1, 3], i=-1
- // x=1, a=[1, 1, 1], i=0

Then, a simple implementation: (tests the tests! also gives us a slow oracle in case we need it later)

```
/**
 * Find the first occurrence of x in sorted array a.
 * @param x value to find
 * @param a array sorted in increasing order (a[0] <= a[1] <= ... <= a[n-
1])
 * @return lowest i such that a[i]==x, or -1 if x not found in a.
 */
public static int find(int x, int[] a) {
    for (int i = 0; i < a.length; ++i) {
        if (x == a[i]) {
            return i;
        }
    }
    return -1;
}
```

Now, some attempts at the real binary-search implementation. Let's do it recursively:

```
public static int search(int x, int[] a) {
    int mid = a.length/2;
    if (x < a[mid]) {
        binarySearch(x, left half of a)
    } else if (x > a[mid]) {
        binarySearch(x, right half of a)
    } else {
        return mid; // because x == a[mid], i.e. we found it!
    }
}
```

OK, we need to strengthen the induction hypothesis to avoid copying the array:

```
/*
 * Find the first occurrence of x in sorted array a[lo..hi].
 * @param x value to find
 * @param a array sorted in increasing order
 *          (a[0] <= a[1] <= ... <= a[n-1])
 * @param lo low end of range.
 *              Requires 0 <= lo <= a.length.
 * @param hi high end of range.
 *              Requires 0 <= hi <= a.length, and lo < hi.
 * @return lowest i such that lo<=i<=hi and a[i]==x,
 *          or -1 if there's no such i.
 */
private static int binarySearchInRange(int x, int[] a, int lo, int hi) {
```

Now:

```
public static int find(int x, int[] a) {
    return binarySearchInRange(x, a, 0, a.length);
}

private static int binarySearchInRange(int x, int[] a, int lo, int hi) {
    int mid = (lo + hi) / 2;
    if (x < a[mid]) {
        return binarySearchInRange(x, a, lo, mid);
    } else if (x > a[mid]) {
        return binarySearchInRange(x, a, mid+1, hi);
    } else {
        // x == a[mid]... we found it!
        return mid;
    }
}
```

Broken! Let's adjust the spec of our private method:

```
/*
 * Find the first occurrence of x in sorted array a[lo..hi-1].
 * @param lo low end of range.
 *              Requires 0 <= lo <= a.length.
 * @param hi high end of range.
 *              Requires 0 <= hi <= a.length, and lo <= hi.
 * @return lowest i such that lo<=i<hi and a[i]==x,
 *          or -1 if there's no such i.
 */
```

```
    private static int binarySearchInRange(int x, int[] a, int lo, int hi)
{...}
    public static int find(int x, int[] a) {
        return binarySearchInRange(x, a, 0, a.length-1);
    }
```

And now add a test for an empty range:

```
    private static int binarySearchInRange(int x, int[] a, int lo, int hi)
{
        if (lo >= hi) {
            return -1; // range has dwindled to nothingness
        }

        int mid = (lo + hi) / 2;
        if (x < a[mid]) {
            return binarySearchInRange(x, a, lo, mid);
        } else if (x > a[mid]) {
            return binarySearchInRange(x, a, mid+1, hi);
        } else {
            // x == a[mid]... we found it!
            return mid;
        }
```

Still broken! Not handling the case of multiple matches correctly. Let's fix:

```
        } else {
            // x == a[mid]... we found it, but check if it's the first
            if (x == a[mid-1]) {
                // not the first! search lower half
                return binarySearchInRange(x, a, lo, mid);
            } else {
                return mid; // it's the first
            }
        }
```

Still broken! In fact, we've experienced a *regression* – tests that used to be passing are now failing because of our change. Regressions happen! Run all your test cases after every change you make!

Need to be careful about a[mid-1], because mid might be 0. Finally:

```
    } else {
            // x == a[mid]... we found it, but check if it's the first
            if (mid > 0 && x == a[mid-1]) {
                // not the first! search lower half
                return binarySearchInRange(x, a, lo, mid);
            } else {
                return mid; // it's the first
            }
    }
```

Here's the final code. Left as an exercise to the reader to tidy up binarySearchInRange() and make it simpler and more compact. The fact that you have a working test suite makes this tidying far safer!

Without a test suite, attempts to simplify logic can easily introduce bugs. Just run it after every little change you make.

```
public static int search(int x, int[] a) {
    return binarySearchInRange(x, a, 0, a.length-1);
}
private static int binarySearchInRange(int x, int[] a, int lo, int hi) {
    if (lo >= hi) {
        return -1; // range has dwindled to nothingness
    }

    int mid = (lo + hi) / 2;
    if (x < a[mid]) {
        return binarySearchInRange(x, a, lo, mid);
    } else if (x > a[mid]) {
        return binarySearchInRange(x, a, mid+1, hi);
    } else {
        // x == a[mid]... we found it, but check if it's the first
        if (mid > 0 && x == a[mid-1]) {
            // not the first! search lower half
            return binarySearchInRange(x, a, lo, mid);
        } else {
            return mid; // it's the first
        }
    }
}
```

Summary

Thinking about our three main measures of code quality (safe from bugs, easy to understand, ready for change), the techniques in this lecture have mainly addressed safety. Readiness for change was considered by writing tests that only depend on behavior in the spec. Testing doesn't measure or improve ease of understanding, but we talked about code review, which does.

testing matters

- you need to convince others (and yourself) that your code works
- testing generally can't prove absence of bugs, but can increase quality by reducing bugs

test early and often

- unit testing catches bugs before they have a chance to hide
- automate the process so you can run it frequently
- regression testing will save time in the long run

be systematic

- use input partitioning, boundary testing, and coverage
- regard testing as a creative design problem

use tools and build your own

- automated testing frameworks (JUnit) and coverage tools (EclEmma)
- design modules to be driven, and use stubs for repeatable behavior

Jwang says like for exponentiation also
do big ints, etc

(why can't they do one for us best practices?)

6.005 L3 Specifications

9/14

- pre conditions
- post conditions
- exceptions
- PS1 due tmo
- PS1 reviewing Fri + Sat
- PS2 out Sat

What this course is about:

Writing code w/ following properties

- safe from bugs
- easy to understand when maintaining / extending
- flexible for change
- fast - not grading on this
  - 6.006
  - 6.172
  - but safe from bugs = more important
- usable
  - 6.813
- security
  - 6.858

(safe from bugs, easy to understand, flexible for change → goals)

(2)

How design interfaces so safe + easy for change

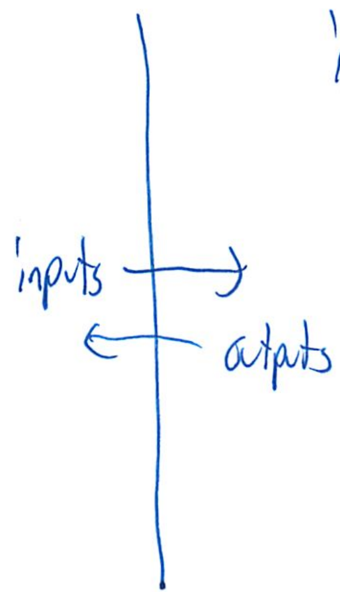
`int find(int x, int [] a)`

method signature

2 different parts

caller (client)

implementor



- requirements on inputs and sometime states

- given good inputs, must return good outputs

Spec is contract between the 2

in classes called interface

(6.004 had something very similar)

- get modularity
  - can replace components

③

Java will statically check for legal inputs

int

int [ ]

But some parts <u>not</u> automatically checked

- like sorting
- need to specify manually

Sidebar

i++ ++i when output not being used - same

i = i + 1
result i

j = i
i = i + 1
result j

when output used
- it matters

Exceptions are good

- know you messed up

Off by 1 elements - can't really notice

(4)

The method sig is not a complete spec

@requires x occurs exactly once in a

@effects int i such that a[i]=x
- if x occurs exactly once

so move this up

If the input is not satisfied in requires
- no promises on what it returns

You want a broader find clause
- so more fully define effects

requires a != null

effects int i s.t. a[i] == x
or -1 if no such i

a could be null
- so put in requires
- but often its implicit

also could confuse output -1
- so say nonnegitive

(5)

We can throw an Exception instead of -1
(object type

```
int find (int x, int[] a) throws NotFoundException
  @requires  true
  @effects   returns i s.t. a[i] = x
             or throws NFE if no such i
```

So two possible return values
- int or NFE
- Java will check it

So when you call, use try statement

```
try {
    find (x, a)
} catch (NFE e) {
    ....
}
```

(6)

Requires ~~clause~~ – implementor is not <u>required to check</u>
– but if cheap to check, do it

Use assert checks to check in junit
– can be turned on or off

A test case should not violate requires clause
– if violates requires, we don't care what it does

If just running for effect

void addAll (List<Str> l1, List<Str> l2)
^ no return value

@requires ~~...~~

@effects ~~...~~ adds eles of l2 to end of l1

@modifies l1

This class will try reduce mutability

What if l1, l2 point to same place
(See this is stuff in Java I don't really know)

So ~~...~~ l1 != l2
^ can't pt to same object

But we could have !l1.eq(l2) & same values

(2)

Java Doc comments
- enforced by human

What is meant by word?
- english
- chinese?

Get precision in specs
- explain word

@return - what happens in tie
- Can leave underdetermined

What if empty?

What if just spaces?

Can have it throw Exception

Implementing

split into words

List easier to work with?

~~Add~~ Count them

- use a Map<String, Integer>

↑ like a dictionary in python

L3: Specifications

Today

- Preconditions & postconditions
- Exceptions

Required reading (from the Java Tutorial)

- Exceptions
- Packages
- Controlling Access

Introduction

In this lecture, we'll look at the role played by specifications of methods. Specifications are the linchpin of teamwork. It's impossible to delegate responsibility for implementing a method without a specification. The specification acts as a contract: the implementer is responsible for meeting the contract, and a client that uses the method can rely on the contract. In fact, we'll see that like real legal contracts, specifications place demands on both parties: when the specification has a pre-condition, the client has responsibilities too.

Why Specifications?

Many of the nastiest bugs in programs arise because of misunderstandings about behavior at interfaces. Although every programmer has specifications in mind, not all programmers write them down. As a result, different programmers on a team have different specifications in mind. When the program fails, it's hard to determine where the error is. Precise specifications in the code let you apportion blame (to code fragments, not people!), and can spare you the agony of puzzling over where a fix should go.

Specifications are good for the client of a method because they spare the task of reading code. If you're not convinced that reading a spec is easier than reading code, take a look at some of the standard Java specs and compare them to the source code that implements them. ArrayList, for example, in the package java.util, has a very simple spec but its code is not at all simple.

Specifications are good for the implementer of a method because they give the freedom to change the implementation without telling clients. Specifications can make code faster too. Sometimes a weak specification makes it possible to do a much more efficient implementation. In particular, a precondition may rule out certain states in which a method might have been invoked that would have incurred an expensive check that is no longer necessary.

The contract acts as a *firewall* between client and implementor. It shields the client from the details of the *workings* of the unit -- you don't need to read the source code of the procedure if you have its specification. And it shields the implementor from the details of the *usage* of the unit; he doesn't have to ask every client how she plans to use the unit. This firewall results in *decoupling*, allowing the code

These lecture notes have been collaboratively authored, with contributions from Saman Amarasinghe, Srini Devadas, Michael Ernst, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. *Copyright © MIT.*

of the unit and the code of a client to be changed independently, so long as the changes respect the specification -- each obeying its obligation.

Behavioral Equivalence

Consider these two methods. Are they the same or different?

```
static int findA (int [] a, int val) {
        for (int i = 0; i < a.length; i++) {
                if (a[i] == val) return i;
        }
        return a.length;
}

static int findB (int [] a, int val) {
        for (int i = a.length -1 ; i >= 0; i--) {
                if (a[i] == val) return i;
        }
        return -1;
}
```

Of course the code is different, so in that sense they are different. Our question though is whether one could substitute one implementation for the other. Not only do these methods have different code; they actually have different behavior:

- when `val` is missing, `findA` returns the length and `findB` returns -1;
- when `val` appears twice, `findA` returns the lower index and `findB` returns the higher.

key, but subtile differences

But when `val` occurs at exactly one index of the array, the two methods behave the same. It may be that clients never rely on the behavior in the other cases. So the notion of equivalence is in the eye of the beholder, that is, the client. In order to make it possible to substitute one implementation for another, and to know when this is acceptable, we need a specification that states exactly what the client depends on.

In this case, our specification might be

```
requires:    val occurs in a
effects:     returns result such that a[result] = val
```

Specification Structure

A specification of a method consists of several clauses:

- a precondition, indicated by the keyword *requires*;
- a postcondition, indicated by the keyword *effects*;
- a frame condition, indicated by the keyword *modifies*.

The precondition is an obligation on the client (i.e., the caller of the method). It's a condition over the state in which the method is invoked. If the precondition does not hold, the implementation of the method is free to do anything (including not terminating, throwing an exception, returning arbitrary results, making arbitrary modifications, etc.).

The postcondition is an obligation on the implementer of the method. If the precondition holds for the invoking state, the method is obliged to obey the postcondition, by returning appropriate values, throwing specified exceptions, modifying or not modifying objects, and so on.

The frame condition is related to the postcondition. It allows more succinct specifications. Without a frame condition, it would be necessary to describe how all the reachable objects may or may not change. But usually only some small part of the state is modifed. The frame condition identifies which objects may be modified. If we say *modifies* x, this means that the object x, which is presumed to be mutable, may be modified, but no other object may be. So in fact, the frame condition or *modifies clause* as it is sometimes called is really an assertion about the objects that are *not* mentioned. For the ones that are mentioned, a mutation is possible but not necessary; for the ones that are not mentioned, a mutation may not occur.

If you omit the frame condition, the default is *modifies nothing*. In other words, the method makes no changes to any object.

Specifications in Java

Some languages (notably Eiffel) incorporate preconditions and postconditions as a fundamental part of the language, as expressions that the runtime system (or possibly even the compiler) can automatically check to enforce the contracts between clients and implementers.

Java does not go quite so far, but its static type declarations *are* effectively part of the precondition and postcondition of a method, a part that is automatically checked and enforced by the compiler. The rest of the contract – the parts that we can't write as types – must be described in a comment preceding the method, and generally depends on human beings to check it and guarantee it.

Java has a convention for documentation comments, in which parameters are described by @param clauses and results are described by @return and @throws clauses. You should generally put the preconditions into @param where possible, and postconditions into @return and @throws. So a specification like this:

```
static int find (int [] a, int val)
requires: val occurs exactly once in a
effects: returns result such that a[result] = val
```

might be rendered in Java like this:

```
/**
 * Find value in an array.
 * @param a array to search; requires that val occurs exactly once in a.
 * @param val value to search for
 * @return index i such that a[i] = val
 */
static int find (int [] a, int val)
```

A frame condition (*modifies* clause) should be described in the @param clauses that it might modify. Side-effects

Find Revisited

Roughly speaking, there are two kinds of specifications. Here is one possible specification of find:

```
static int find (int [] a, int val)
requires: val occurs exactly once in a
effects: returns result such that a[result] = val
```

This specification is deterministic: when presented with a state satisfying the precondition, the outcome is determined. Both findA and findB satisfy the specification, so if this is the specification

on which the clients relied, the two are equivalent and substitutable for one another. (Of course a procedure must have the *name* demanded by the specification; here we are using different names to allow us to talk about the two versions. To use either, you'd have to change its name to find.)

Here is a slightly different specification:

```
static int find (int [] a, int val)
      requires: val occurs in a
      effects: returns result such that a[result] = val
```

This specification is not deterministic. Such a specification is often said to be non-deterministic, but this is a bit misleading. Non-deterministic code is code that you expect to sometimes behave one way and sometimes another. This can happen, for example, with concurrency: the scheduler chooses to run threads in different orders depending on conditions outside the program.

But a 'non-deterministic' specification doesn't call for such non-determinism in the code. The behavior specified is not non-deterministic but *under-determined*. In this case, the specification doesn't say which index is returned if val occurs more than once; it simply says that if you look up the entry at the index given by the returned value, you'll find val.

This specification is again satisfied by both findA and findB, each 'resolving' the underdeterminedness in its own way. A client of find can't predict which index will be returned, but should not expect the behavior to be truly non-deterministic. Of course, the specification is satisfied by a non-deterministic procedure too -- for example, one that rather improbably tosses a coin to decide whether to start searching from the top or the bottom of the array. But in almost all cases we'll encounter, non-determinism in specifications offers a choice that is made by the implementor at implementation time, and not at runtime.

So, as before, for this specification too, the two versions of find are equivalent. Finally, here's a specification that distinguishes the two

```
static int find (int [] a, int val)
      effects: returns largest result such that
               a[result] = val or -1 if no such result
```

It is satisfied by findB but not findA.

Specification for a Mutating Method

Our specifications of find didn't give us the opportunity to illustrate frame conditions and the description of side effects.

Here's a specification that describes a method that mutates an object:

```
static boolean addAll (List l1, List l2)
      requires: l1 != l2, l1!=null, l2!=null
      modifies: this
      effects: adds the elements of l2 to the end of l1,
               and returns true if l1 changed as a
            result of call
```

We've taken this, slightly simplified, from the Java List interface. First, look at the frame condition: it tells us that only this is modified, so in particular the argument *l2* is not mutated -- likely to be a crucial property for most clients. Second, look at the postcondition. It gives two constraints: the first

telling us how this is modified, and the second telling us how the return value is determined. Finally, look at the precondition. It tells us that the behavior of the method is not constrained if you call it with a null argument, or if you attempt to add the elements of a list to itself. The constraint that arguments not be null is often left implicit in practice, but you can easily imagine why the implementor of the method would want to impose the second constraint: it's not likely to rule out any useful applications of the method, and it makes it easier to implement. The specification allows a simple implementation in which you take an element from *l2* and add it to *l1*, then go on to the next element of *l1* until you get to the end. If *l1* and *l2* are the same list, this algorithm will not terminate -- an outcome permitted by the specification.

Declarative Specification

Roughly speaking, there are two kinds of specifications. *Operational* specifications give a series of steps that the method performs; pseudocode descriptions are operational. *Declarative* specifications don't give details of intermediate steps. Instead, they just give properties of the final outcome, and how it's related to the initial state.

Almost always, declarative specifications are preferable. They're usually shorter, easier to understand, and most importantly, they don't expose implementation details inadvertently that a client may rely on (and then find no longer hold when the implementation is changed). For example, if we want to allow either implementation of `find`, we would not want to say in the spec that the method "goes down the array until it finds `val`," since aside from being rather vague, this spec suggests that the search proceeds from lower to higher indices and that the lowest will be returned, which perhaps the specifier did not intend.

Specification Ordering

Suppose you want to substitute one method for another. How do you compare the specifications?

A specification **A** is at least as strong as a specification **B** if

- **A**'s precondition is no stronger than **B**'s
- **A**'s postcondition is no weaker than **B**'s, for the states that satisfy B's precondition.

These two rules embody several ideas. They tell you that you can always weaken the precondition; placing fewer demands on a client will never upset him. You can always strengthen the postcondition, which means making more promises. For example, our method `maybePrime` can be replaced in any context by a method `isPrime` that returns true if and only if the integer is prime. And where the precondition is false, you can do whatever you like. If the postcondition happens to specify the outcome for a state that violates the precondition, you can ignore it, since that outcome is not guaranteed anyway.

Judging Specifications

What makes a good method? Designing a method means primarily writing a specification. There are no infallible rules, but there are some useful guidelines. About the form of the specification: it should obviously be succinct, clear and well-structured. The content is harder to prescribe.

The specification should be *coherent*: it shouldn't have lots of different cases. Long argument lists, deeply nested if-statements, and boolean flags are a sign of trouble. Consider this specification:

```
static int minFind (int[] a, int[] b, int val)
      effects: returns smallest index in arrays a and b at which
               val appears
```

Is this a well-designed procedure? Probably not: it's incoherent, since it does two things (finding and minimizing) that are not really related. It would be better to use two separate procedures.

The results of a call should be *informative*. Consider the specification of a method that puts a value in a map:

```
static V put (Map<K,V> map, K key, V val)
        effects: inserts (key, val) into the mapping,
                 overriding any existing mapping for key, and
                 returns old value for key, unless none,
                 in which case it returns null
```

Note that the precondition does not rule out null values, so the map can store nulls. But the postcondition uses null as a special return value for a missing key. This means that if null is returned, you can't tell whether the key was not bound previously, or whether it was in fact bound to null. This is not a very good design, because the return value is useless unless you know you didn't insert nulls.

The specification should be *strong enough*. There's no point throwing a checked exception for a bad argument but allowing arbitrary mutations, because a client won't be able to determine what mutations have actually been made. Here's a specification illustrating this flaw (and also written in an inappropriately operational style):

```
static void addAll (List<T> l1, List<T> l2)
        effects: adds the elements of l2 to l1,
                  unless it encounters a null element,
                  at which point it throws a NullPointerException
```

The specification should be *weak enough*. Consider this specification for a method that opens a file:

```
static File open (String filename)
        effects: opens a file named filename
```

This is a bad specification. It lacks important details: is the file opened for reading or writing? Does it already exist or is it created? And it's too strong, since there's no way it can guarantee to open a file. The process in which it runs may lack permission to open a file, or there might be some problem with the file system beyond the control of the program. Instead, the specification should say something much weaker: that it attempts to open a file, and if it succeeds, the file has certain properties.

Exceptions for Special Results

You've probably already seen some exceptions in your Java programming so far, such as ArrayOutOfBoundsException (thrown when an array index a[i] is outside the valid range for the array) or NullPointerException (thrown when trying to call a method on a null object reference).

But exceptions are not just for handling failures like these. They can be used to improve the structure of code that involves procedures with special results.

A standard way to handle special results is to return special values. Lookup operations in the Java library are often designed like this: you get an index of -1 when expecting a positive integer, or a null reference when expecting an object. This approach is OK if used sparingly. It has two problems though. First, it's tedious to check the return value. Second, it's easy to forget to do it. (We'll see that with exceptions you can get help from the compiler in this.)

Also, its not easy to find a 'special value'. Suppose we have a `BirthdayBook` class with a lookup method. Here's one possible method signature:

```
Calendar lookup (String name)
```

What should the method do if the birthday book doesn't have an entry for the person whose name is given? Well, we could return some special date that is not going to be used as a real date. Bad programmers have been doing this for decades; they would return 9/9/99, for example, since it was *obvious* that no program written in 1960 would still be running at the end of the century.

Here's a better approach. The method throws an exception:

```
Calendar lookup (String name) throws NotFoundException {
  ...
  if // not found
    throw new NotFoundException ();
  ...
```

and the caller handles the exception with a catch clause. Now there's no need for any special value, nor the checking associated with it.

Abuse of Exceptions

Here's an example from *Effective Java* by Joshua Bloch (Item 39).

```
try {
   int i = 0;
   while (true)
       a[i++].f();
} catch(ArrayIndexOutofBoundsException e) { }
```

What does this code do? It is not at all obvious from inspection, and that's reason enough not to use it. The infinite loop terminates by throwing, catching and ignoring an `ArrayIndexOutofBoundsException` when it attempts to access the first array element outside the bounds of the array. It is supposed to be equivalent to:

```
for(int i = 0; i < a.length; i++)
   a[i].f();
```

The exception-based idiom is a misguided attempt to improve performance based on the faulty reasoning that, since the VM checks the bounds of array accesses, the normal loop termination test (`i < a.length`) is redundant and should be avoided. However, because exceptions in Java are designed for use only under exceptional circumstances, few, if any, JVM implementations attempt to optimize their performance. On a typical machine, the exception-based idiom runs 70 times slower than the standard one when looping from 0 to 99.

The exception-based idiom is also not guaranteed to work. Suppose the computation of `f()` in the body of the loop contains a bug that results in an out-of-bounds access to some unrelated array. If a reasonable loop idiom were used, the bug would generate an uncaught exception, resulting in immediate thread termination with an appropriate error message. If the exception-based idiom were used, the bug-related exception would be caught and misinterpreted as a normal loop termination.

Checked and Unchecked Exceptions

We've seen two different purposes for exceptions: failures and special results. Java provides two different kinds of exception for these two purposes. They behave the same at runtime; the only difference is what kind of checking the compiler provides.

If a method might throw a *checked* exception, the possibility must be declared in its signature. `NotFoundException` would be a checked exception, and that's why the signature ends `throws NotFoundException`. If a method calls another method that may throw a checked exception, it must either handle it, or declare the exception itself (since if it isn't caught locally it will be propagated).

So if you call the `lookup` method and forget to handle the exception, the compiler will reject your code. This is very useful, because it ensures that exceptions that are expected to occur should be handled. On the other hand, exceptions that correspond to failures are not expected to be handled except at the top level, and for reasons of modularity, we wouldn't want to declare the possibility of failure at every level.

For an *unchecked* exception, in contrast, the compiler will not check for try-catch or a throws declaration. Java still allows you write a throws clause as part of a signature for an unchecked exception, but this has no effect (and is thus a bit funny, and I don't recommend doing it).

All errors and exceptions may have a message associated with them. If not provided in the constructor, the reference to the message string is null.

Exceptions and Preconditions

An obvious design issue is whether to use a precondition, and if so, whether it should be checked. It is crucial to understand that a precondition does not require that checking be performed. On the contrary, the most common use of preconditions is to demand a property precisely because it would be hard or expensive to check.

As mentioned above, a non-trivial precondition renders the method partial. This inconveniences clients, because they have to ensure that they don't call the method in a bad state (that violates the precondition); if they do, there is no predictable way to recover from the error. So users of methods don't like preconditions, and for this reason the methods of a library will usually be total. That's why the Java API classes, for example, invariably throw exceptions when arguments are inappropriate. It makes the programs in which they are used more robust.

Sometimes, it's not feasible to check a condition without making a method unacceptably slow, and a precondition is often necessary in this case. If we wanted to implement the find() method using binary search, we would have to require that the array be sorted. Forcing the method to actually *check* that the array is sorted would defeat the entire purpose of the binary search: to obtain a result in logarithmic and not linear time.

The decision of whether to use a precondition is an engineering judgment. The key factors are the cost of the check (in writing and executing code), and the scope of the method. If it's only called locally in a class, the precondition can be discharged by carefully checking all the sites that call the method. But if the method is public, and used by other developers, it would be less wise to use a precondition.

Design Considerations

The rule we have given -- use checked exceptions for special results, and unchecked exceptions to signal failures -- makes sense, but it isn't the end of the story. The snag is that exceptions in Java aren't as lightweight as they might be.

Aside from the performance penalty, exceptions in Java incur another (more serious) cost: they're a pain to use. If you design a method to have its own exception, you have to create a new class for the exception. If you call a method that can throw a checked exception, you have to wrap it in a `try-catch` statement (even if you know the exception will never be thrown). This latter stipulation creates a dilemma. Suppose, for example, you're designing a queue abstraction. Should popping the queue throw a checked exception when the queue is empty? Suppose you want to support a style or programming in the client in which the queue is popped (in a loop say) until the exception is thrown. So you choose a checked exception. Now some client wants to use the method in a context in which, immediately prior to popping, the client tests whether the queue is empty and only pops if it isn't. Maddeningly, that client will still need to wrap the call in a `try-catch` statement.

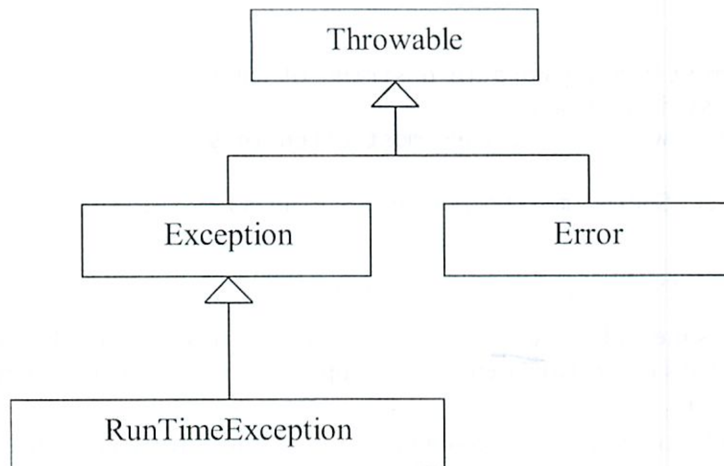
This suggests a more refined rule:

- You should use an unchecked exception only if you expect that clients will usually write code that ensures the exception will not happen, because there is a convenient and inexpensive way to avoid the exception, or because the exception reflects unexpected failures;
- Otherwise you should use a checked exception.

The cost of using exceptions in Java is one reason that many Java API's use the null reference as a special value. It's not a terrible thing to do, so long as it's done judiciously, and carefully specified.

Throwable Hierarchy

Let's look at the class hierarchy for Java exceptions.



`Throwable` is the class of objects that can be thrown or caught. Any object used in a `throw` or `catch` statement, or declared in the `throws` clause of a method, must be a subclass of `Throwable`. `Throwable`'s implementation records a stack trace at the point where the exception was thrown, along with an optional string describing the exception.

`Error` is a subclass of `Throwable` that is "reserved" for errors produced by the Java runtime system, such as `StackOverflowError` and `OutofMemoryError`. For some reason `AssertionError` also extends `Error`, even though it indicates a failure in user code, not in Java runtime. Errors should be considered recoverable, and are generally not caught. All the unchecked throwables you implement should subclass `RunTimeException` (directly or indirectly).

The classes `Error` and `RuntimeException` correspond to unchecked exceptions. All other `Throwables` and `Exceptions` are checked. Do not define a throwable that is not a subclass of `Exception`, `RunTimeException`, or `Error`.

A word about access control

We have been using *public* for almost all of our methods, without really thinking about it. The decision to make a method public or private is actually a decision about the contract of the class. Public methods are freely accessible to other parts of the program. Making a method public advertises it as a service that your class is willing to provide. If you make all your methods public – including helper methods that are really meant only for local use within the class – then other parts of the program may come to depend on them, which will make it harder for you to change the internal implementation of the class in the future. Your code won't be as **ready for change**.

Making internal helper methods public will also add clutter to the visible interface your class offers. Keeping internal things *private* makes your class's public interface smaller and more coherent (meaning that it does one thing and does it well). Your code will be **easier to understand**.

We will see even strong reasons to use *private* in the next few lectures, when we start to write classes with persistent internal state. Protecting this state will help keep the program **safe from bugs**.

An example of specification design

Let's finish by working an example in which we design some specifications. Suppose we're given this spec to implement:

```
/**
 * Find the most common word in a string of text.
 * @param s string of words
 * @return the word that occurs most often in s
 */
public static String mostCommonWord(String s) { ... }
```

Let's push back and criticize the spec we've been given.

- what exactly is meant by a "word?" the spec should be precise about this; that's not a kind of freedom that's particularly useful for the implementer, and it makes the method hard to use for the client.
- how are words compared, when counting their frequency? the spec should be precise about that too.
- the spec fails to even touch on two important cases. It only discusses the case where there is exactly one most common word, and doesn't put requirements on the client or the implementer to deal with the other cases. That means neither the client or the implementer will do it! And bugs will follow. So we need to think about the "0" case – what if there are no words in the string at all? – and about the "many" case – what if there are ties for most-common word? – and ensure that these cases are addressed either by preconditions or postconditions.

Here's how we might handle both these issues with a stronger postcondition:

```
/**
* Find the most common word in a string of text.
* @param s string of words, where a word is
*   a nonempty string of characters separated by spaces
```

```
 *    or punctuation.
 * @return a word that occurs most often in s (at least as much
 *      as any other word).  Alphabetic case is ignored when comparing
 *      words.
 * @throws NoWordException if s has no words
 */
public static String mostCommonWord(String s)
        throws NoWordsException { ... }
```

We've handled the "many answers" case with wording: the return value will be a word that occurs "at least as much as any other word," leaving the implementer free to decide how to break ties. That freedom will be convenient for us; if we wrote a more precise specification, such as the word that occurs first in s should win in the event of a tie, we might have to do a lot more bookkeeping inside mostCommonWord, remembering where each word was.

To handle the "no answers" case, we've introduced a new kind of exception, NoWordsException. We'll have to declare that exception class in order to use it. Here's the simplest way to do that:

```
/**
 * Exception thrown by mostCommonWord() when it can't find a word.
 */
public static class NoWordsException extends Exception {
}
```

Now we're ready to think about implementing this method. Let's break this problem down into several smaller steps:

1. Split the string into words.
2. Count the number of occurrences of each word.
3. Find the word with the maximum count.

We'll write each of these steps as a private helper method, so we can think about them as individual units (that we could test individually[1]).

Let's start with the method that splits the string into words. First we'll write the *signature* for the method, which has the types of its arguments and return values:

```
private static List<String> splitIntoWords(String s) {...}
```

We might have returned a String[] here instead of a List, but lists are easier to work with. Now we'll write its spec:

```
// Split s into words.
// @param s string to split
// @return list of words found in s, in order of their occurrence
//  (where word is defined by the spec for mostCommonWords).
private static List<String> splitIntoWords(String s) { ... }
```

The next step takes this list of strings and counts its elements. We'll write its spec this way:

```
// Count the number of occurrences of each element in a list.
// @param l list of strings
// @return map m such that m[s] == k if s occurs k times in l, while
//      m[s] == null if s never occurs in l.
private static Map<String, Integer> countOccurrences(List<String> l) {
    throw new RuntimeException("not implemented");
```

[1] Note that JUnit can't easily test private methods, so to use JUnit for testing these methods, we'd need to work around it.

```
}
```

The return type we chose here is a Map, which is a data structure that maps values of one type (the *keys*, in this case Strings) into values of another type (the map's *values*, here Integers, or effectively ints). The spec says that every element of the input list becomes a key in the map, and the value corresponding to it is the number of times it appears in the list.

Note the decision here not even to talk about words, but more generally about "elements of the list." Taking the extra step to think and write more generally is often a good thing, because it makes the code (and its spec) easier to change in the future. If someday we wanted to change the caller of countOccurrences – perhaps to pass in lists of phrases, not just words – then the greater flexibility in the spec will be useful. This should be weighed carefully against other considerations, however, and it's not a good idea to make code more complex just to make it very general.

Now let's look at the combination of splitIntoWords and countOccurrences. We'll probably want to plug them together like this:

```
List<String> words = splitIntoWords(s);
Map<String, Integer> freq = countOccurrences(words);
```

But a requirement of mostCommonWord's contract has fallen through the cracks! Who's responsible for the ignore-alphabetic-case obligation? We could insert a new method in between that lower-cases the list of words. Or we could put the onus on either splitIntoWords or countOccurrences to handle that obligation. Let's strengthen the spec of splitIntoWords:

```
// Split s into words.
// @param s string to split
// @return list of words found in s, in order of their occurrence
//  (where word is defined by the spec for mostCommonWords).
//  The words are all converted to lowercase.
private static List<String> splitIntoWords(String s) { ... }
```

The final step of our process needs a method to find the word that has maximum count:

```
// Find a key with maximum value.
// @param m frequency counts for strings
// @return s such that m[s] >= m[t] for all other keys t in the map,
//     or null if no such s exists
private static String findMax(Map<String,Integer> m) {...}
```

Note that we have written the postcondition of this method in a declarative way. We also chose to return null rather than throw an exception in the case where the map is empty. Is this a good idea or bad idea?

Summary

A specification acts as a crucial firewall between the implementor of a procedure and its client. It makes separate development possible: the client is free to write code that uses the procedure without seeing its source code, and the implementor is free to write the code that implements the procedure without knowing how it will be used. Declarative specifications are the most useful in practice. Preconditions make life hard for the client, but, applied judiciously, are a vital tool in the software designer's repertoire.

Home Ask a Question Jobs Projects Forum Submit Tutorial Latest Tutorials Subscribe in a reader

Search

Integer: byte, short, int, and long data types in Java

By: Abinaya Emailed: 243 times Printed: 362 times 1

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. Many other computer languages, including C/C++, support both signed and unsigned integers. However, Java's designers felt that unsigned integers were unnecessary.

Specifically, they felt that the concept of *unsigned* was used mostly to specify the behavior of the *high-order bit,* which defined the *sign* of an **int** when expressed as a number. Java manages the meaning of the high-order bit differently, by adding a special "unsigned right shift" operator. Thus, the need for an unsigned integer type was eliminated.

The *width* of an integer type should not be thought of as the amount of storage it consumes, but rather as the *behavior* it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them. In fact, at least one implementation stores **bytes** and **shorts** as 32-bit (rather than 8- and 16-bit) values to improve performance, because that is the word size of most computers currently in use.

The width and ranges of these integer types vary widely, as shown in this table:

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9 ,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	– 32,768 to 32,767
byte	8	– 128 to 127

Let's look at each type of integer.

byte

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 to 127. Variables of type **byte** are especially useful when you're working with a stream of data from a network or file. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types. Byte variables are declared by use of the **byte** keyword. For example, the following declares two **byte** variables called **b** and **c**:

```
byte b, c;
```

short

short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type, since it is defined as having its high byte first (called *big-endian* format). This type is mostly applicable to 16-bit computers, which are becoming increasingly scarce. Here are some examples of **short** variable declarations:

```
short s;
short t;
```

Android
AJAX
ASP.ne
C
Cocoa
C++
C#
Java
Javasc
JSF
JSP
J2ME
Java Be
JDBC
Linux
Mac OS
Perl
PHP
Python
Ruby
VB.net
Hiberna
Struts
Projects
Trends
Certific
Intervie
Site Ma
Contact
Forum

Note
"Endianness" describes how multibyte data types, such as **short**, **int**, and **long**, are stored in memory. If it takes 2 bytes to represent a **short**, then which one comes first, the most significant or the least significant? To say that a machine is big-endian, means that the most significant byte is first, followed by the least significant one. Machines such as the SPARC and PowerPC are big-endian, while the Intel x86 series is little-endian.

int

The most commonly used integer type is **int**. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays. Any time you have an integer expression involving **bytes, shorts, ints,** and literal numbers, the entire expression is *promoted* to **int** before the calculation is done.

The **int** type is the most versatile and efficient type, and it should be used most of the time when you want to create a number for counting or indexing arrays or doing integer math. It may seem that using **short** or **byte** will save space, but there is no guarantee that Java won't promote those types to **int** internally anyway. Remember, type determines behavior, not size. (The only exception is arrays, where **byte** is guaranteed to use only one byte per array element, **short** will use two bytes, and **int** will use four.)

long

long is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value. The range of a **long** is quite large. This makes it useful when big, whole numbers are needed. For example, here is a program that computes the number of miles that light will travel in a specified number of days.

```
// Compute distance light travels using long variables.
class Light {
public static void main(String args[]) {
int lightspeed;
long days;
long seconds;
long distance;
// approximate speed of light in miles per second
lightspeed = 186000;
days = 1000; // specify number of days here
seconds = days * 24 * 60 * 60; // convert to seconds
distance = lightspeed * seconds; // compute distance
System.out.print("In " + days);
System.out.print(" days light will travel about ");
System.out.println(distance + " miles.");
}
}
```

This program generates the following output:

In 1000 days light will travel about 16070400000000 miles.

Clearly, the result could not have been held in an **int** variable.

If this tutorial doesn't answer your question, and you have a specific question, just ask an expert here. Post your question to get a direct answer.

Related

Find us o

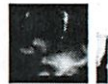
Java-Sa

Like

1,744 peop
Java-Samp

Habiba

Bala



Aashish P

Facebook

Tutorial A
Android
C Cocoa
Java Ce
Interview
JSF JSP
J2ME JD
OS X Pe
Ruby VB
Struts Hi

Latest Tu
Trends A
ASP.net
C# Java
Interview
JSF JSP
J2ME JD
OS X Pe
Ruby VB
Struts Hi

Get tuto
your em

Su

Add PS1

9/14

Add more test cases
- diff int sizes
- null for objects

- no real detailed list either in notes or reading way
- In his demo he did not do anything crazy

- I put one in at int boundry
  - fails
  - answer seems to be not the modulus ...

2147483647^2 mod 5

just do

a $\cdot$ a mod m

So needs to be long
- but insists on int

(2)

$v(t) = 2.5t - 450$

↑ oh duh want to $= 0$ - include both parts

~~[illegible]~~ 8.01 style!

$x(t) = 1.25t^2 - 450t + x_0$

↑ ~~what is~~

what is x_0 - when/<u>what height</u> activate rockets

$t = $ ~~[illegible]~~ $\frac{450}{2.5} = 180$ s

$x_0 = 0 - (1.25)(180)^2 + 450(180) = 40{,}500$

So tricked - no diff eq at all!!!

I *think* I would get it if I thought more about it

Really worked out - start

- goal
- etc

6.005 Recitation

9/15

Q: Power Mod long
- my Pset

Various data structures

Exhaustive things to test for

Specification

- contract between callers + implementation
- requires acceptable inputs
  preconditions
  - ie list must be sorted
    int must be positive
- output acceptable outputs / effects
- modifies - side effects

```
/**  [description]
  *  @params
  *  @return
 **/
```

(2)

The caller does not care about the exact details of the implementation

---

Q what to test for

Special case

0

$-\infty$ ← is in int class -

Max_int

Q: Increment Map

```
Map<String, Integer> m = new HashMap<String, Integer>();
m.put("hello", 5)
m.get("hello")++;
```

Q: knowing all these data structures

types

Objects

primitives

learn them

Objects somewhat more memory

(I think I just need to be more careful + do next P-Set in lab)

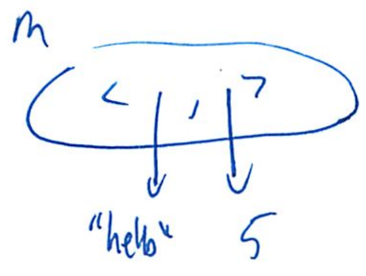
③

Can put day in a map

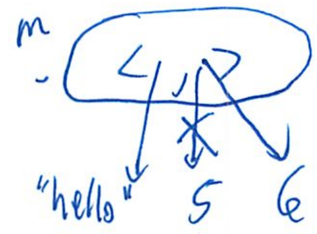
```
Map <Sting, Date> m = new HashMap <Sting, Date>();
m.put("hello", new Date.(2011, 9,14));
m.get("hello".increase Day(5));
```

? can do this
since object

but Integer is immutable
- so can't change



then change



Now work on division code

(4)

- put in what you do w/ special cases
  b != 0
- what returning int or double, etc
- how you will round/truncate

Specification ordering

One is stronger than others

A is as least as strong as B if
- A's precondition is no stronger than B's
- B's post condition is no stronger than A's
- A is not making asumptions B hasn't

6.005 PS1 More

9/15

Org missed 6.005 Tue recitation - not on calendar as well
- but those are not really worth it

Adding a lot test cased ~~crashed~~ program other one to work
And similar one crashed program
- well caused it to go ∞

Ending at a certain i for whatever reason
So TA showed me new way
- 3rd way to do it
~~And~~ That's all I care to fix
Just wanted to perfect that 1

6.005 Review PS1

9/17

$+=$ takes $O(n^2)$ time

- new string object each time

Use StringBuilder instead

(More fun to go last - after other people have reviewed!)

Can use .contains

? all these things I don't realize!

Hard to adjust function to function

- need to remember/review my code

I'm writing all my comments w/ "I"

- since un sure of standards

Only do 1 function/test

For letters Main.BASIC_ALPHABET

②

I should have tested out of order training data!

Java Data Structures (2nd edition)

© 1996-2001, Particle

Introduction...

Welcome to ***Java Data Structures*** (2nd edition). This document was created with an intent to show people how easy Java really is, and to clear up a few *things* I've missed in the previous release of the document.

This is a growing document; as new features are added to the language, new techniques are discovered or realized, this document shall be updated to try to accommodate them all. If you have suggestions, or requests, (or spelling/grammar errors) just e-mail them, and I'll try to add the suggested topics into the subsequent release. Because this document is changing so much, I've decided to implement a version number. This release is: *v2.2.11*, updated: *May 7th, 2002*.

Current release of the document, including all the sources, can be downloaded here:

[download zip with sources and everything]

Of course, this document is free, and I intend to keep it that way. Selling of this document is NOT permitted. You *WILL* go to hell if you do (*trust me*). (not that anybody would want to buy it...) You may distribute this document (in *ANY* form), provided you don't change it. (yes, you CAN include it in a book provided you notify me and give me credit <and give me one free copy of the book>) To my knowledge, this document has already been reproduced and distributed within some corporations, schools and colleges, but has yet to be formally published in a book.

I take no responsibility for *ANYTHING*. I am only responsible for all the good things you like about the article. So, remember, if it's bad, don't blame me, if it's good, thank me (give me credit).

All the source has been compiled and tested using *JDK v1.2*. Although most things should work flawlessly with previous versions, there are things where *JDK 1.2* is more appropriate. If you find problems and/or errors, please let me know.

Although this document should be read in sequence, it is divided into several major sections, here they are:

Variables

Arrays
Array Stack
Array Queue
Array List
The Vector

Nodes

Linked Lists
Reusing Tricks

Trees
Generic Tree
Comparing Objects
Binary Search Trees
Tree Traversals

Node Pools
Node Pool Nodes
Node Pool Generic Trees
Node Pool Sort Trees

Priority Vectors
Sorting
Sorting JDK 1.2 Style
Sorting using Quicksort
Optimizing Quicksort
Radix Sort

Improving Radix Sort

Reading and Writing Trees (Serialization)
Deleting items from a Binary Search Tree
Determining Tree Depth

Advanced Linked Lists
Doubly Linked Lists (with Enumeration)

Binary Space Partition Trees (BSP)
Binary Space Partition Tree DEMO (Dog 3D)
Binary Space Partition Tree DEMO with Lighting (Dog 3D)

Kitchen Sink Methods
Java Native Interface (JNI)

Bibliography
Special Thanks
Contact Info

In contrast to what most people think about Java, it being a language with no pointers, data structures are quite easy to implement. In this section, I'll demonstrate few basic data structures. By learning how easy they are to implement in Java, you'll be able to write any implementation yourself.

I also think that this document is a pretty good introduction to ***Data Structures*** in general. All these concepts can be applied in any programming language. Incidentally, most of these programs are ported from their C++ counterparts. So, if you want to learn Data Structures in C/C++, you'll still find this document useful! Java is an Object Oriented language, but more so than C++, so, most data structure concepts are expressed and illustrated *"more naturally"* in Java! (*try not to raise your blood pressure from all the caffeine*)

I suggest that you be familiar with Java format, and know some other programming language in advance. Coincidentally, I and a couple of my friends are in the process of writing a C language book, which deals with all that "start up" stuff.

The way most examples are executed is through the JDK's command line Java interpreter. (at the prompt, you just type "`java`" and the name of the class to run.)

Variables...

Variables are the key to any program. There are variables called registers inside every CPU (Central Processing Unit). Every program ever written uses some form of variables. Believe it or not, the way you use variables can significantly impact your program. This section is a very simple introduction to what variables are, and how they're used in programs.

Usually, a variable implies a memory location to hold one instance of one specific type. What this means is that if there is an integer variable, it can only hold one integer, and if there is a character variable, it can only hold one character.

There can be many different types of variables, including of your own type. A sample declaration for different variable types is given below.

```
boolean t;
byte b;
char c;
int i;
long l;
```

I believe the above is straight forward, and doesn't need much explanation. Variable `'t'` is declared as `boolean` type, and `'b'` as of `byte` type, etc.

The above variables are what's know as primitive types. Primitive types in Java means that you don't have to create them, they're already available as soon as you declare them. (you'll see what I mean when we deal with Objects) It also means that there is usually some hardware equivalent to these variables. For example, an `int` type, can be stored in a 32 bit hardware register.

The other types of variables are instances of classes or Objects. Java is a very *Object Oriented* language, and everything in it, is an object. An object is an instance of a class. Your Java programs consist of classes, in which you manipulate objects, and make the whole program do what you want. This concept will be familiar to you if you've ever programmed C++, if not, think of objects as structures. An example of a simple class would be:

```
public class pSimpleObject{
    int i;
    public pSimpleObject(){
        i = 0;
    }
    public int get(){
        return i;
    }
    public void set(int n){
        i = n;
    }
}
```

As you can see, first we specify that the class is `public`, this means that it can be visible to other objects outside it's file. We later say that it's a `class`, and give it's name, which in this case is: `pSimpleObject`. Inside of it, the class contains an integer named `'i'`, and three functions. The first function named `pSimpleObject()`, is the constructor. It is called every time an object is created using this class. The `set()` and `get()` functions set and get the value of `'i'` respectively. One useful terminology is that functions in objects are not called functions, they're called methods. So, to refer to function `set()`, you'd say "method `set()`." That's all there is to objects!

The way you declare a variable, or in this case, an object of that class, is:

```
pSimpleObject myObject;
myObject = new pSimpleObject();
```

or

```
pSimpleObject myObject = new pSimpleObject();
```

The first example illustrates how you declare an object named `myObject`, of class `pSimpleObject`, and later instantiate it (a process of actual creation, where it calls the object's constructor method). The second approach illustrates that it all can be done in one line. The object does not get created when you just declare it, it's only created when you do a `new` on it.

If you're familiar with C/C++, think of objects as pointers. First, you declare it, and then you allocate a new object to that pointer. The only limitation seems to be that you can't do math on these pointers, other than that, they behave as plain and simple C/C++ pointers. (You might want to think of objects as references however.)

Using variables is really cool, and useful, but sometimes we'd like to have more. Like the ability to work with hundreds or maybe thousands of variables at the same time. And here's where our next section starts, the Arrays!

Arrays...

One of the most basic data structures, is an array. An array is just a number of items, of same type, stored in linear order, one after another. Arrays have a set limit on their size, they can't grow beyond that limit. Arrays usually tend to be easier to work with and generally more efficient than other structural approaches to organizing data; way better than a *no formal structure* approach.

For example, lets say you wanted to have 100 numbers. You can always resort to having 100 different variables, but that would be a pain. Instead, you can use the clean notation of an array to create, and later manipulate those 100 numbers. For example, to create an array to hold 100 numbers you would do something like this:

```
int[] myArray;
myArray = new int[100];
```

or

```
int[] myArray = new int[100];
```

or

```
int myArray[] = new int[100];
```

The three notations above do exactly the same thing. The first declares an array, and then it creates an array by doing a `new`. The second example shows that it can all be one in one line. And the third example shows that Java holds the backwards compatibility with C++, where the array declaration is: `int myArray[];` instead of `int[] myArray;`. To us, these notations are exactly the same. I do however prefer to use the Java one.

Working with arrays is also simple, think of them as just a line of variables, we can address the 5th element (counting from 0, so, it's actually the 6th element) by simply doing:

```
int i = myArray[5];
```

The code above will set integer 'i' to the value of the 5th (counting from 0) element of the array. Similarly, we can set an array value. For example, to set the 50th element (counting from 0), to the value of 'i' we'd do something like:

```
myArray[50] = i;
```

As you can see, arrays are fairly simple. The best and most convenient way to manipulate arrays is using loops. For example, lets say we wanted to make an array from 1 to 100, to hold numbers from 1 to 100 respectively, and later add seven to every element inside that array. This can be done very easily using two loops. (actually, it can be done in one loop, but I am trying to separate the problem into two)

```
int i;
for(i=0;i<100;i++)
    myArray[i] = i;
for(i=0;i<100;i++)
    myArray[i] = myArray[i] + 7;
```

In Java, we don't need to remember the size of the array as in C/C++. Here, we have the length variable in every array, and we can check it's length whenever we need it. So to print out any array named: myArray, we'd do something like:

```
for(int i = 0;i<myArray.length;i++)
    System.out.println(myArray[i]);
```

This will work, given the objects inside the myArray are printable, (have a corresponding toString() method), or are of primitive type.

One of the major limitations on arrays is that they're fixed in size. They can't grow or shrink according to need. If you have an array of 100 max elements, it will not be able to store 101 elements. Similarly, if you have less elements, then the unused space is being wasted (doing nothing).

Java API provides data storage classes, which implement an array for their storage. As an example, take the java.util.Vector class (JDK 1.2), it can grow, shrink, and do some quite useful things. The way it does it is by reallocating a new array every time you want to do some of these operations, and later copying the old array into the new array. It can be quite fast for small sizes, but when you're talking about several megabyte arrays, and every time you'd like to add one more number (or object) you might need to reallocate the entire array; that can get quite slow. Later, we will look at other data structures where we won't be overly concerned with the amount of the data and how often we need to resize.

Even in simplest situations, arrays are powerful storage constructs. Sometimes, however, we'd like to have more than just a plain vanilla array.

Array Stack...

The next and more serious data structure we'll examine is the Stack. A stack is a FILO (First In, Last Out), structure. For now, we'll just deal with the array representation of the stack. Knowing that we'll be using an array, we automatically think of the fact that our stack has to have a maximum size.

A stack has only one point where data enters or leaves. We can't insert or remove elements into or from the middle of the stack. As I've mentioned before, everything in Java is an object, (since it's an Object Oriented language), so, lets write a stack object!

```
public class pArrayStackInt{
    protected int head[];
    protected int pointer;

    public pArrayStackInt(int capacity){
        head = new int[capacity];
        pointer = -1;
    }
    public boolean isEmpty(){
        return pointer == -1;
    }
    public void push(int i){
        if(pointer+1 < head.length)
            head[++pointer] = i;
    }
    public int pop(){
        if(isEmpty())
            return 0;
        return head[pointer--];
    }
}
```

As you can see, that's the stack class. The constructor named `pArrayStackInt()` accepts an integer. That integer is to initialize the stack to that specific size. If you later try to `push()` more integers onto the stack than this capacity, it won't work. Nothing is complete without testing, so, lets write a test driver class to test this stack.

```
import java.io.*;
import pArrayStackInt;

class pArrayStackIntTest{
    public static void main(String[] args){
        pArrayStackInt s = new pArrayStackInt(10);
        int i,j;
        System.out.println("starting...");
        for(i=0;i<10;i++){
            j = (int)(Math.random() * 100);
            s.push(j);
            System.out.println("push: " + j);
        }
        while(!s.isEmpty()){
            System.out.println("pop: " + s.pop());
        }
        System.out.println("Done ;-)");
    }
}
```

The test driver does nothing special, it inserts ten random numbers onto the stack, and then pops them off. Writing to standard output exactly what it's doing. The output gotten from this program is:

```
starting...
push: 33
push: 66
push: 10
push: 94
push: 67
push: 79
push: 48
push: 7
push: 79
push: 32
pop: 32
pop: 79
pop: 7
pop: 48
pop: 79
pop: 67
pop: 94
pop: 10
pop: 66
pop: 33
Done ;-)
```

As you can see, the first numbers to be pushed on, are the last ones to be popped off. A perfect example of a FILO structure. The output also assures us that the stack is working properly.

Now that you've had a chance to look at the source, lets look at it more closely.

The `pArrayStackInt` class is using an array to store it's data. The data is `int` type (for simplicity). There is a head data member, that's the actual array. Because we're using an array, with limited size, we need to keep track of it's size, so that we don't overflow it; we always look at `head.length` to check for maximum size.

The second data member is `pointer`. Pointer, in here, points to the top of the stack. It always has the position which had the last insertion, or -1 if the stack is empty.

The constructor: `pArrayStackInt()`, accepts the maximum size parameter to set the size of the stack. The rest of the functions is just routine initialization. Notice that pointer is initialized to -1, this makes the next position to be filled in an array, 0.

The `isEmpty()` function is self explanatory, it returns `true` if the stack is empty (`pointer` is -1), and `false` otherwise. The return type is `boolean`.

The `push(int)` function is fairly easy to understand too. First, it checks to see if the next insertion will not overflow the array. If no danger from overflow, then it inserts. It first increments the pointer and then inserts into the new location pointed to by the updated `pointer`. It could easily be modified to actually make the array grow, but then the whole point of "simplicity" of using an array will be lost.

The `int pop()` function is also very simple. First, it checks to see if stack is not empty, if it is empty, it will return 0. In general, this is a really bad error to pop of something from an empty stack. You may want to do something more

sensible than simply returning a 0 (an exception throw would not be a bad choice). I did it this way for the sake of simplicity. Then, it returns the value of the array element currently pointed to by pointer, and it decrements the pointer. This way, it is ready for the next `push` or `pop`.

I guess that just about covers it. Stack is very simple and is very basic. There are tons of useful algorithms which take advantage of this FILO structure. Now, lets look at alternative implementations...

Given the above, a lot of the C++ people would look at me strangely, and say: "All this trouble for a stack that can only store integers?" Well, they're probably right for the example above. It is too much trouble. The trick I'll illustrate next is what makes Java my favorite Object Oriented language.

In C, we have the `void*` type, to make it possible to store "generic" data. In C++, we also have the `void*` type, but there, we have very useful templates. Templates is a C++ way to make generic objects, (objects that can be used with any type). This makes quite a lot of sense for a data storage class; why should we care what we're storing?

The way Java implements these kinds of generic classes is by the use of parent classes. In Java, every object is a descendent of the `Object` class. So, we can just use the `Object` class in all of our structures, and later cast it to an appropriate type. Next, we'll write an example that uses this technique inside a generic stack.

```
public class pArrayStackObject{
    protected Object head[];
    protected int pointer;

    public pArrayStackObject(int capacity){
        head = new Object[capacity];
        pointer = -1;
    }
    public boolean isEmpty(){
        return pointer == -1;
    }
    public void push(Object i){
        if(pointer+1 < head.length)
            head[++pointer] = i;
    }
    public Object pop(){
        if(isEmpty())
            return null;
        return head[pointer--];
    }
}
```

The above is very similar to the `int` only version, the only things that changed are the `int` to `Object`. This stack, allows the `push()` and `pop()` of any `Object`. Lets convert our old test driver to accommodate this new stack. The new test module will be inserting `java.lang.Integer` objects (not `int`; not primitive type).

```
import java.io.*;
import pArrayStackObject;

class pArrayStackObjectTest{
    public static void main(String[] args){
        pArrayStackObject s = new pArrayStackObject(10);
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<10;i++){
            j = new Integer((int)(Math.random() * 100));
            s.push(j);
            System.out.println("push: " + j);
        }
        while(!s.isEmpty()){
            System.out.println("pop: " + ((Integer)s.pop()));
        }
        System.out.println("Done ;-)");
    }
}
```

And for the sake of being complete, I'll include the output. Notice that here, we're not inserting elements of `int` type, we're inserting elements of `java.lang.Integer` type. This means, that we can insert any `Object`.

```
starting...
push: 45
push: 7
push: 33
push: 95
push: 28
push: 98
```

```
push: 87
push: 99
push: 66
push: 40
pop: 40
pop: 66
pop: 99
pop: 87
pop: 98
pop: 28
pop: 95
pop: 33
pop: 7
pop: 45
Done ;-)
```

I guess that covers stacks. The main idea you should learn from this section is that a stack is a FILO data structure. After this section, non of the data structures will be working with primitive types, and everything will be done solely with objects. (now that you know how it's done...)

And now, onto the array relative of Stack, the Queue.

Array Queues...

A queue is a FIFO (First In, First Out) structure. Anything that's inserted first, will be the first to leave (kind of like the real world queues.) This is totally the opposite of what a stack is. Although that is true, the queue implementation is quite similar to the stack one. It also involves pointers to specific places inside the array.

With a queue, we need to maintain two pointers, the `start` and the `end`. We'll determine when the queue is empty if `start` and `end` point to the same element. To determine if the queue is full (since it's an array), we'll have a `boolean` variable named `full`. To insert, we'll add one to the `start`, and mod (the `%` operator) with the size of the array. To remove, we'll add one to the `end`, and mod (the `%` operator) with the size of the array. Simple? Well, lets write it.

```
public class pArrayQueue{
    protected Object[] array;
    protected int start,end;
    protected boolean full;

    public pArrayQueue(int maxsize){
        array = new Object[maxsize];
        start = end = 0;
        full = false;
    }

    public boolean isEmpty(){
        return ((start == end) && !full);
    }

    public void insert(Object o){
        if(!full)
            array[start = (++start % array.length)] = o;
        if(start == end)
            full = true;
    }

    public Object remove(){
        if(full)
            full = false;
        else if(isEmpty())
            return null;
        return array[end = (++end % array.length)];
    }
}
```

Well, that's the queue class. In it, we have four variables, the `array`, the `start` and `end`, and a `boolean full`. The constructor `pArrayQueue(int maxsize)` initializes the queue, and allocates an `array` for data storage. The `isEmpty()` method is self explanatory, it checks to see if `start` and `end` are equal; this can only be in two situations: when the queue is empty, and when the queue is full. It later checks the `full` variable and returns whether this queue is empty or not.

The `insert(Object)` method, accepts an `Object` as a parameter, checks whether the queue is not `full`, and inserts it. The insert works by adding one to `start`, and doing a mod with `array.length` (the size of the `array`), the resulting location is set to the incoming object. We later check to see if this insertion caused the queue to become full, if yes, we note this by setting the `full` variable to `true`.

The `Object remove()` method, doesn't accept any parameters, and returns an `Object`. It first checks to see if the queue is `full`, if it is, it sets `full` to `false` (since it will not be `full` after this removal). If it's not `full`, it checks if the queue is empty, by calling `isEmpty()`. If it is, the method returns a `null`, indicating that there's been an error. This is usually a pretty bad bug inside a program, for it to try to remove something from an empty queue, so, you might want to do something more drastic in such a situation (like an exception throw). The method continues by removing the end object from the queue. The removal is done in the same way insertion was done. By adding one to the `end`, and later mod it with `array.length` (`array` size), and that position is returned.

There are other implementations of the same thing, a little re-arrangement can make several `if()` statements disappear. The reason it's like this is because it's pretty easy to think of it. Upon insertion, you add one to `start` and mod, and upon removal, you add one to `end` and mod... easy?

Well, now that we know how it works, lets actually test it! I've modified that pretty cool test driver from the stack example, and got it ready for this queue, so, here it comes:

```
import java.io.*;
import pArrayQueue;

class pArrayQueueTest{
    public static void main(String[] args){
        pArrayQueue q = new pArrayQueue(10);
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<10;i++){
            j = new Integer((int)(Math.random() * 100));
            q.insert(j);
            System.out.println("insert: " + j);
        }
        while(!q.isEmpty()){
            System.out.println("remove: " + ((Integer)q.remove()));
        }
        System.out.println("Done ;-)");
    }
}
```

As you can see, it inserts ten random `java.lang.Integer Objects` onto the queue, and later prints them out. The output from the program follows:

```
starting...
insert: 3
insert: 70
insert: 5
insert: 17
insert: 26
insert: 79
insert: 12
insert: 44
insert: 25
insert: 27
remove: 3
remove: 70
remove: 5
remove: 17
remove: 26
remove: 79
remove: 12
remove: 44
remove: 25
remove: 27
Done ;-)
```

I suggest you compare this output to the one from stack. It's almost completely different. I guess that's it for this array implementation of this FIFO data structure. And now, onto something more complex...

Array Lists...

The next step up in complexity is a list. Most people prefer to implement a list as a linked list (and I'll show how to do that later), but what most people miss, is that lists can also be implemented using arrays. A list has no particular structure; it just has to allow for the insertion and removal of objects from both ends, and some way of looking at the middle elements.

A list is kind of a stack combined with a queue; with additional feature of looking at the middle elements. Preferably, a

list should also contain the current number of elements. Well, lets not just talk about a list, but write one!

```
public class pArrayList{
    protected Object[] array;
    protected int start,end,number;

    public pArrayList(int maxsize){
        array = new Object[maxsize];
        start = end = number = 0;
    }
    public boolean isEmpty(){
        return number == 0;
    }
    public boolean isFull(){
        return number >= array.length;
    }
    public int size(){
        return number;
    }
    public void insert(Object o){
        if(number < array.length){
            array[start = (++start % array.length)] = o;
            number++;
        }
    }
    public void insertEnd(Object o){
        if(number < array.length){
            array[end] = o;
            end = (--end + array.length) % array.length;
            number++;
        }
    }
    public Object remove(){
        if(isEmpty())
            return null;
        number--;
        int i = start;
        start = (--start + array.length) % array.length;
        return array[i];
    }
    public Object removeEnd(){
        if(isEmpty())
            return null;
        number--;
        return array[end = (++end % array.length)];
    }
    public Object peek(int n){
        if(n >= number)
            return null;
        return array[(end + 1 + n) % array.length];
    }
}
```

The class contains four data elements: `array`, `start`, `end`, and `number`. The `number` is the number of elements inside the array. The `start` is the starting pointer, and the `end` is the ending pointer inside the `array` (kind of like the queue design).

The constructor, `pArrayList()`, and methods `isEmpty()`, `isFull()`, and `size()`, are pretty much self explanatory. The `insert()` method works exactly the same way as an equivalent queue method. It just increments the `start` pointer, does a mod (the `%` symbol), and inserts into the resulting position.

The `insertEnd(Object)` method, first checks that there is enough space inside the `array`. It then inserts the element into the `end` location. The next trick is to decrement the `end` pointer, add the `array.length`, and do a mod with `array.length`. This had the effect of moving the `end` pointer backwards (as if we had inserted something at the end).

The `Object remove()` method works on a very similar principle. First, it checks to see if there are elements to remove, if not, it simply returns a `null` (no `Object`). It then decrements `number`. We're keeping track of this `number` inside all insertion and removal methods, so that it always contains the current `number` of elements. We then create a temporary variable to hold the current position of the `start` pointer. After that, we update the `start` pointer by first decrementing it, adding `array.length` to it, and doing a mod with `array.length`. This gives the appearance of removing an element from the front of the list. We later return the position inside the array, which we've saved earlier inside that temporary variable `'i'`.

The `Object removeEnd()` works similar to the `insert()` method. It checks to see if there are elements to remove by calling `isEmpty()` method, if there aren't, it returns `null`. It then handles the `number` (number of elements) business, and proceeds with updating the `end` pointer. It first increments the `end` pointer, and then does a mod with `array.length`,

and returns the resulting position. Simple?

This next `Object peek(int n)` method is the most tricky one. It accepts an integer, and we need to return the number which this integer is pointing to. This would be no problem if we were using an `array` that started at `0`, but we're using our own implementation, and the list doesn't necessarily start at `array` position `0`. We start this by checking if the parameter `'n'` is not greater than the `number` of elements, if it is, we return `null` (since we don't want to go past the bounds of the `array`). What we do next is add `'n'` (the requesting number) to an incremented `end` pointer, and do a mod `array.length`. This way, it appears as if this function is referencing the array from `0` (while the actual start is the incremented `end` pointer).

As I've said previously, the code you write is useless, unless it's working, so, lets write a test driver to test our list class. To write the test driver, I've converted that really cool Queue test driver, and added some features to test out the specifics of lists. Well, here it is:

```
import java.io.*;
import pArrayList;

class pArrayListTest{
    public static void main(String[] args){
        pArrayList l = new pArrayList(10);
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<5;i++){
            j = new Integer((int)(Math.random() * 100));
            l.insert(j);
            System.out.println("insert: " + j);
        }
        while(!l.isFull()){
            j = new Integer((int)(Math.random() * 100));
            l.insertEnd(j);
            System.out.println("insertEnd: " + j);
        }
        for(i=0;i<l.size();i++)
            System.out.println("peek "+i+": "+l.peek(i));
        for(i=0;i<5;i++)
            System.out.println("remove: " + ((Integer)l.remove()));
        while(!l.isEmpty())
            System.out.println("removeEnd: " + ((Integer)l.removeEnd()));
        System.out.println("Done ;-)");
    }
}
```

The test driver is nothing special, it inserts (in front) five random numbers, and the rest into the back (also five). It then prints out the entire list by calling `peek()` inside a `for` loop. It then continues with the removal (from front) of five numbers, and later removing the rest (also five). At the end, the program prints "Done" with a cute smiley face ;-)

The output from this test driver is given below. I suggest you examine it thoroughly, and make sure you understand what's going on inside this data structure.

```
starting...
insert: 14
insert: 72
insert: 71
insert: 11
insert: 27
insertEnd: 28
insertEnd: 67
insertEnd: 36
insertEnd: 19
insertEnd: 45
peek 0: 45
peek 1: 19
peek 2: 36
peek 3: 67
peek 4: 28
peek 5: 14
peek 6: 72
peek 7: 71
peek 8: 11
peek 9: 27
remove: 27
remove: 11
remove: 71
remove: 72
remove: 14
removeEnd: 45
```

```
removeEnd: 19
removeEnd: 36
removeEnd: 67
removeEnd: 28
Done ;-)
```

Well, if you really understand everything up to this point, there is nothing new anybody can teach you about arrays (since you know all the basics). There are however public tools available to simplify your life. Some are good, some are bad, but one that definitely deserves to have a look at is the `java.util.Vector` class; and that's what the next section is about!

The Vector...

The `java.util.Vector` class is provided by the Java API, and is one of the most useful array based data storage classes I've ever seen. The information provided here is as far as JDK 1.2 goes, future versions may have other implementations; still, the functionality should remain the same. A vector, is a growing array; as more and more elements are added onto it, the array grows. There is also a possibility of making the array smaller.

But wait a minute, all this time I've been saying that arrays can't grow or shrink, and it seems Java API has done it. Not quite. The `java.util.Vector` class doesn't exactly grow, or shrink. When it needs to do these operations, it simply allocates a new array (of appropriate size), and copies the contents of the old array into the new array. Thus, giving the impression that the array has changed size.

All these memory operations can get quite expensive if a `Vector` is used in a wrong way. Since a `Vector` has a similar architecture to the array stack we've designed earlier, the best and fastest way to implement a `Vector` is to do stack operations. Usually, in programs, we need a general data storage class, and don't really care about the order in which things are stored or retrieved; that's where `java.util.Vector` comes in very useful.

Using a `Vector` to simulate a queue is very expensive, since every time you insert or remove, the entire array has to be copied (not necessarily reallocated but still involves lots of useless work).

`Vector` allows us to view it's insides using an `Enumerator`; a class to go through objects. It is very useful to first be able to look what you're looking for, and only later decide whether you'd like to remove it or not. A sample program that uses `java.util.Vector` for it's storage follows.

```
import java.io.*;
import java.util.*;

class pVectorTest{
    public static void main(String[] args){
        Vector v = new Vector(15);
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<10;i++){
            j = new Integer((int)(Math.random() * 100));
            v.addElement(j);
            System.out.println("addElement: " + j);
        }
        System.out.println("size: "+v.size());
        System.out.println("capacity: "+v.capacity());

        Enumeration enum = v.elements();
        while(enum.hasMoreElements())
            System.out.println("enum: "+(Integer)enum.nextElement());

        System.out.println("Done ;-)");
    }
}
```

The example above should be self evident (if you paid attention when I showed test programs for the previous data structures). The main key difference is that this one doesn't actually remove objects at the end; we just leave them inside. Removal can be accomplished very easily, and if you'll be doing anything cool with the class, you'll sure to look up the API specs.

Printing is accomplished using an `Enumerator`; which we use to march through every element printing as we move along. We could also have done the same by doing a `for` loop, going from `0` to `v.size()`, doing a `v.elementAt(int)` every time through the loop. The output from the above program follows:

```
starting...
addElement: 9
addElement: 5
addElement: 54
```

```
addElement: 49
addElement: 60
addElement: 81
addElement: 8
addElement: 91
addElement: 76
addElement: 81
size: 10
capacity: 15
enum: 9
enum: 5
enum: 54
enum: 49
enum: 60
enum: 81
enum: 8
enum: 91
enum: 76
enum: 81
Done ;-)
```

You should notice that when we print the `size` and `capacity`, they're different. The `size` is the current number of elements inside the `Vector`, and the `capacity`, is the maximum possible without reallocation.

A trick you can try yourself when playing with the `Vector` is to have `Vectors` of `Vectors` (since `Vector` is also an `Object`, there shouldn't be any problems of doing it). Constructs like that can lead to some interesting data structures, and even more confusion. Just try inserting a `Vector` into a `Vector` ;-)

I guess that covers the `Vector` class. If you need to know more about it, you're welcome to read the API specs for it. I also greatly encourage you to look at `java.util.Vector` source, and see for yourself what's going on inside that incredibly simple structure.

Nodes...

The other type of data structures are what's called Node based data structures. Instead of storing data in it's raw format, Node based data structures store nodes, which in turn store the data. Think of nodes as being elements, which may have one or more pointers to other nodes.

Yes, I did say the "pointer" word. Many people think that there are no pointers in Java, but just because you don't see them directly, doesn't mean they're not there. In fact, you can treat any object as a pointer.

Thus, the Node structure should have a data element, and a reference to another node (or nodes). Those other nodes which are referenced to, are called child nodes. The node itself is called the parent node (or sometimes a "father" node) in reference to it's children. (nice big happy family)

Well, the best way to visualize a node is to create one, so, lets do it. The node we'll create will be a one child node (it will have only one pointer), and we'll later use it in later sections to build really cool data structures. The source for our one child node follows:

```
public class pOneChildNode{
    protected Object data;
    protected pOneChildNode next;

    public pOneChildNode(){
        next = null;
        data = null;
    }
    public pOneChildNode(Object d,pOneChildNode n){
        data = d;
        next = n;
    }
    public void setNext(pOneChildNode n){
        next = n;
    }
    public void setData(Object d){
        data = d;
    }
    public pOneChildNode getNext(){
        return next;
    }
    public Object getData(){
        return data;
    }
    public String toString(){
```

```
        return ""+data;
    }
}
```

Go over the source, notice that it's nothing more than just set and get functions (pretty simple). The two data members are the `data` and `next`. The `data` member holds the data of the node, and `next` holds the pointer to the next node. Notice that `next` is of the same type as the class itself; it effectively points to the object of same class!

The `String toString()` method is the Java's standard way to print things. If an object wants to be printed in a special way, it will define this method, with instructions on how to print this object. In our case, we just want to print the `data`. Adding `data` to a bunch of quotation marks automatically converts it to type `String` (hopefully, our data will also have a `toString()` method defined on it). Without this method, we get the actual binary representation of the data members of this class (not a pretty nor meaningful printout).

Node based data structures provide for dynamic growing and shrinking, and are the key to some complex algorithms (as you'll see later). Now that we know how to implement a Node, lets get to something cool...

Linked Lists...

A linked list is just a chain of nodes, with each subsequent node being a child of the previous one. Many programs rely on linked lists for their storage because these don't have any evident restrictions. For example, the array list we did earlier could not grow or shrink, but node based ones can! This means there is no limit (other than the amount of memory) on the number of elements they can store.

A linked list has just one node, that node, has links to subsequent nodes. So, the entire list can be referenced from that one node. That first node is usually referred to as the head of the list. The last node in the chain of nodes usually has some special feature to let us know that it's last. That feature, most of the time is a `null` pointer to the next node.

```
[node0]->[node1]->[node2]->[node3]->[node4]->null
```

The example above illustrates the node organization inside the list. In it, `node0` is the head node, and `node4` is the last node, because it's pointer points to `null`. Well, now that you know how it's done, and what is meant by a linked list, lets write one. (I mean, that's why we're here, to actually write stuff!)

```
import pOneChildNode;

public class pLinkedList{
    protected pOneChildNode head;
    protected int number;

    public pLinkedList(){
        head = null;
        number = 0;
    }
    public boolean isEmpty(){
        return head == null;
    }
    public int size(){
        return number;
    }
    public void insert(Object obj){
        head = new pOneChildNode(obj,head);
        number++;
    }
    public Object remove(){
        if(isEmpty())
            return null;
        pOneChildNode tmp = head;
        head = tmp.getNext();
        number--;
        return tmp.getData();
    }
    public void insertEnd(Object obj){
        if(isEmpty())
            insert(obj);
        else{
            pOneChildNode t = head;
            while(t.getNext() != null)
                t=t.getNext();
            pOneChildNode tmp =
                new pOneChildNode(obj,t.getNext());
            t.setNext(tmp);
            number++;
        }
```

```
    }
    public Object removeEnd(){
        if(isEmpty())
            return null;
        if(head.getNext() == null)
            return remove();
        pOneChildNode t = head;
        while(t.getNext().getNext() != null)
            t = t.getNext();
        Object obj = t.getNext().getData();
        t.setNext(t.getNext().getNext());
        number--;
        return obj;
    }
    public Object peek(int n){
        pOneChildNode t = head;
        for(int i = 0;i<n && t != null;i++)
            t = t.getNext();
        return t.getData();
    }
}
```

← So this implements Child node data structure

Before we move on, lets go over the source. There are two data members, one named `head`, and the other named `number`. Head is the first node of the list, and `number` is the total number of nodes in the list. Number is primarily used for the `size()` method. The constructor, `pLinkedList()` is self explanatory. The `size()` and `isEmpty()` methods are also pretty easy.

Here comes the hard part, the insertion and removal methods. Method `insert(Object)` creates a new `pOneChildNode` object with `next` pointer pointing to the current `head`, and `data` the data which is inserted. It then sets the `head` to that new node. If you think about it, you'll notice that the `head` is still saved, and the new node points to it.

Method `Object remove()` works in a very similar fashion, but instead of inserting, it is removing. It first checks to see if the list is `isEmpty()` or not, if it is, it returns a `null`. It then saves the current `head` node, and then changes it to accommodate the removal (think about the logic), decrements the `number`, and returns the `data` from the previously saved node.

In the method `insertEnd(Object)`, we're actually inserting at the end of the list. We first check to see if the list is `isEmpty()`, if it is, we do a regular insertion (since it really doesn't matter which direction we're coming from if the list is `empty`). We then setup a loop to search for the end. The end is symbolized by the `next` pointer of the node being `null`. When we get to the end, we create a new node, and place it at the end location. Incrementing `number` before we return.

Method `Object removeEnd()` works in a similar fashion as `insertend(Object)` method. It also goes through the whole list to look for the end. At the beginning, we check if the list is not `isEmpty()`, if it is, we return a `null`. We then check to see if there is only one element in the list, if there is only one, we remove it using regular `remove()`. We then setup a loop to look for the node who's child is the last node. It is important to realize that if we get to the last node, we won't be able to erase it; we need the last node's parent node. When we find it, we get the `data`, setup necessary links, decrement `number`, and return the `data`.

The `Object peek(int)` method simply goes through the list until it either reaches the element requested, or the end of the list. If it reaches the end, it should return a `null`, if not, it should return the correct location inside the list.

As I have said before, it is very important to actually test. The ideas could be fine, and logical, but if it doesn't work, it doesn't work. So, lets convert our `pArrayListTest` driver to accommodate this class.

all this stuff in CS I never knew

```
import java.io.*;
import pLinkedList;

class pLinkedListTest{
    public static void main(String[] args){
        pLinkedList l = new pLinkedList();
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<5;i++){
            j = new Integer((int)(Math.random() * 100));
            l.insert(j);
            System.out.println("insert: " + j);
        }
        for(;i<10;i++){
            j = new Integer((int)(Math.random() * 100));
            l.insertEnd(j);
            System.out.println("insertEnd: " + j);
        }
        for(i=0;i<l.size();i++)
            System.out.println("peek "+i+": "+l.peek(i));
```

```
        for(i=0;i<5;i++)
            System.out.println("remove: " + ((Integer)l.remove()));
        while(!l.isEmpty())
            System.out.println("removeEnd: " + ((Integer)l.removeEnd()));
        System.out.println("Done ;-)");
    }
}
```

The test driver is nothing special, it's just a pretty simple conversion of the old test driver, so, I wont spend any time discussing it. The output follows.

```
starting...
insert: 65
insert: 78
insert: 21
insert: 73
insert: 62
insertEnd: 82
insertEnd: 63
insertEnd: 6
insertEnd: 95
insertEnd: 57
peek 0: 62
peek 1: 73
peek 2: 21
peek 3: 78
peek 4: 65
peek 5: 82
peek 6: 63
peek 7: 6
peek 8: 95
peek 9: 57
remove: 62
remove: 73
remove: 21
remove: 78
remove: 65
removeEnd: 57
removeEnd: 95
removeEnd: 6
removeEnd: 63
removeEnd: 82
Done ;-)
```

Look over the output, make sure you understand why you get what you get. Linked lists are one of the most important data structures you'll ever learn, and it really pays to know them well. Don't forget that you can always experiment. One exercise I'd like to leave up to the reader is to create a circular list. In a circular list, the last node is not pointing to `null`, but to the first node (creating a circle). Sometimes, lists are also implemented using two pointers; and there are many other variations you should consider and try yourself. You can even make it faster by having a "dummy" first node and/or "tail" node. This will eliminate most special cases, making it faster on insertions and deletions.

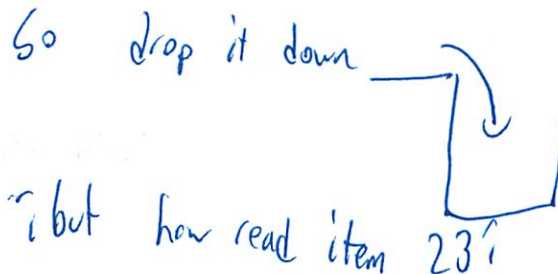
I guess that's it for the lists, next, I'll show you how to do simple and easy tricks to re-use code.

Reusing Tricks...

We have already written quite a lot of useful stuff, and there might come a time, when you're just too lazy to write something new, and would like to reuse the old source. This section will show you how you can convert some data structures previously devised, to implement a stack and a queue, with almost no creativity (by simply reusing the old source).

Before we start, lets review the function of a stack. It has to be able to push and pop items of from one end. What structure do we know that can do something similar? A list! Lets write a list implementation of a stack.

```
import pLinkedList;

public class pEasyStack{
    protected pLinkedList l;

    public pEasyStack(){
        l = new pLinkedList();
    }
    public boolean isEmpty(){
        return l.isEmpty();
    }
    public void push(Object o){
```



```
        l.insert(o);
    }
    public Object pop(){
        return l.remove();
    }
}
```

See how easily the above code simulates a stack by using a list? It may not be the best implementation, and it's certainly not the fastest, but when you need to get the project compiled and tested, you don't want to spend several unnecessary minutes writing a full blown optimized stack. Test for the stack follows:

```
import java.io.*;
import pEasyStack;

class pEasyStackTest{
    public static void main(String[] args){
        pEasyStack s = new pEasyStack();
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<10;i++){
            j = new Integer((int)(Math.random() * 100));
            s.push(j);
            System.out.println("push: " + j);
        }
        while(!s.isEmpty()){
            System.out.println("pop: " + ((Integer)s.pop()));
        }
        System.out.println("Done ;-)");
    }
}
```

The stack test program is exactly the same as for the previous stack version (doesn't really need explanation). For the completion, I'll also include the output.

```
starting...
push: 23
push: 99
push: 40
push: 78
push: 54
push: 27
push: 52
push: 34
push: 98
push: 89
pop: 89
pop: 98
pop: 34
pop: 52
pop: 27
pop: 54
pop: 78
pop: 40
pop: 99
pop: 23
Done ;-)
```

You've seen how easily we can make a stack. What about other data structures? Well, we can just as easily implement a queue. One thing though, now instead of just inserting and removing, we'll be removing from the end (the other from the one we're inserting).

```
import pLinkedList;

public class pEasyQueue{
    protected pLinkedList l;

    public pEasyQueue(){
        l = new pLinkedList();
    }
    public boolean isEmpty(){
        return l.isEmpty();
    }
    public void insert(Object o){
        l.insert(o);
    }
    public Object remove(){
        return l.removeEnd();
```

```
    }
}
```

Pretty easy huh? Well, the test driver is also easy.

```
import java.io.*;
import pEasyQueue;

class pEasyQueueTest{
    public static void main(String[] args){
        pEasyQueue s = new pEasyQueue();
        Integer j = null;
        int i;
        System.out.println("starting...");
        for(i=0;i<10;i++){
            j = new Integer((int)(Math.random() * 100));
            s.insert(j);
            System.out.println("insert: " + j);
        }
        while(!s.isEmpty()){
            System.out.println("remove: " + ((Integer)s.remove()));
        }
        System.out.println("Done ;-)");
    }
}
```

I guess you get the picture, reusing code may not always be the best choice, but it sure is the easiest! Definitely, if you have time, always write a better implementation; these approaches are only good for the deadline, just to compile and test the code before the actual hard work of optimizing it. For the sake of completeness, the output of the above program follows:

```
starting...
insert: 77
insert: 79
insert: 63
insert: 59
insert: 22
insert: 62
insert: 54
insert: 58
insert: 79
insert: 25
remove: 77
remove: 79
remove: 63
remove: 59
remove: 22
remove: 62
remove: 54
remove: 58
remove: 79
remove: 25
Done ;-)
```

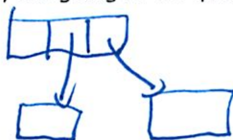
I guess that covers code reuse! And remember, you can also reuse the code in the Java API. The reuse can be of many forms. You can reuse it the way I've shown in this section, or you can extend that particular class to provide the necessary functions. (Extending a `java.util.Vector` class is very useful ;-)

Trees...

The next major set of data structures belongs to what's called Trees. They are called that, because if you try to visualize the structure, it kind of looks like a tree (root, branches, and leafs). Trees are node based data structures, meaning that they're made out of small parts called nodes. You already know what a node is, and used one to build a linked list. Tree Nodes have two or more child nodes; unlike our list node, which only had one child.

Trees are named by the number of children their nodes have. For example, if a tree node has two children, it is called a binary tree. If it has three children, it is called tertiary tree. If it has four children, it is called a quad tree, and so on. Fortunately, to simplify things, we only need binary trees. With binary trees, we can simulate any tree; so the need for other types of trees only becomes a matter of simplicity for visualization. well not really

Since we'll be working with binary trees, lets write a binary tree node. It's not going to be much different from our `pOneChildNode` class; actually, it's going to be quite the same, only added support for one more pointer. The source for the follows:



Such a little change makes huge difference

```
public class pTwoChildNode{
    protected Object data;
    protected pTwoChildNode left,right;

    public pTwoChildNode(){
        data = null;
        left = right = null;
    }
    public pTwoChildNode(Object d){
        data = d;
        left = right = null;
    }
    public void setLeft(pTwoChildNode l){
        left = l;
    }
    public void setRight(pTwoChildNode r){
        right = r;
    }
    public void setData(Object d){
        data = d;
    }
    public pTwoChildNode getLeft(){
        return left;
    }
    public pTwoChildNode getRight(){
        return right;
    }
    public Object getData(){
        return data;
    }
    public String toString(){
        return ""+data;
    }
}
```

This node is so much similar to the previous node we did, that I'm not even going to cover this one. (I assume you'll figure it out by simply looking at the source) The children of the node are named `left` and `right`; these will be the left branch of the node and a right branch. If a node has no children, it is called a leaf node. If a node has no parent (it's the father of every node), it is called the root of the tree. This weird terminology comes from the tree analogy, and from the family tree analogy.

Some implementations of the tree node, might also have a back pointer to the parent node, but for what we'll be doing with the nodes, it's not necessary. The next section will talk about a generic binary tree which will be later used to create something cool.

Generic Tree...

Binary Trees are quite complex, and most of the time, we'd be writing a unique implementation for every specific program. One thing that almost never changes though is the general layout of a binary tree. We will first implement that general layout as an abstract class (a class that can't be used directly), and later write another class which extends our layout class.

Trees have many different algorithms associated with them. The most basic ones are the traversal algorithms. Traversals algorithms are different ways of going through the tree (the order in which you look at it's values). For example, an in-order traversal first looks at the left child, then the data, and then the right child. A pre-order traversal, first looks at the data, then the left child, and then the right; and lastly, the post-order traversal looks at the left child, then right child, and only then data. Different traversal types mean different things for different algorithms and different trees. For example, if it's binary search tree (I'll show how to do one later), then the in-order traversal will print elements in a sorted order.

Well, lets not just talk about the beauties of trees, and start writing one! The code that follows creates an abstract Generic Binary Tree.

```
import pTwoChildNode;

public abstract class pGenericBinaryTree{
    private pTwoChildNode root;

    protected pTwoChildNode getRoot(){
        return root;
    }
    protected void setRoot(pTwoChildNode r){
        root = r;
    }
```

```
    public pGenericBinaryTree(){
        setRoot(null);
    }
    public pGenericBinaryTree(Object o){
        setRoot(new pTwoChildNode(o));
    }
    public boolean isEmpty(){
        return getRoot() == null;
    }
    public Object getData(){
        if(!isEmpty())
            return getRoot().getData();
        return null;
    }
    public pTwoChildNode getLeft(){
        if(!isEmpty())
            return getRoot().getLeft();
        return null;
    }
    public pTwoChildNode getRight(){
        if(!isEmpty())
            return getRoot().getRight();
        return null;
    }
    public void setData(Object o){
        if(!isEmpty())
            getRoot().setData(o);
    }
    public void insertLeft(pTwoChildNode p,Object o){
        if((p != null) && (p.getLeft() == null))
            p.setLeft(new pTwoChildNode(o));
    }
    public void insertRight(pTwoChildNode p,Object o){
        if((p != null) && (p.getRight() == null))
            p.setRight(new pTwoChildNode(o));
    }
    public void print(int mode){
        if(mode == 1) pretrav();
        else if(mode == 2) intrav();
        else if(mode == 3) postrav();
    }
    public void pretrav(){
        pretrav(getRoot());
    }
    protected void pretrav(pTwoChildNode p){
        if(p == null)
            return;
        System.out.print(p.getData()+" ");
        pretrav(p.getLeft());
        pretrav(p.getRight());
    }
    public void intrav(){
        intrav(getRoot());
    }
    protected void intrav(pTwoChildNode p){
        if(p == null)
            return;
        intrav(p.getLeft());
        System.out.print(p.getData()+" ");
        intrav(p.getRight());
    }
    public void postrav(){
        postrav(getRoot());
    }
    protected void postrav(pTwoChildNode p){
        if(p == null)
            return;
        postrav(p.getLeft());
        postrav(p.getRight());
        System.out.print(p.getData()+" ");
    }
}
```

Now, lets go over it. The `pGenericBinaryTree` is a fairly large class, with a fair amount of methods. Lets start with the one and only data member, the `root`! In this `abstract class`, `root` is a `private` head of the entire tree. Actually, all we need is `root` to access anything (and that's how you'd implement it in other languages). Since we'd like to have access to `root` from other places though (from derived classes, but not from the "outside," we've also added two methods, named `getRoot()`, and `setRoot()` which get and set the value of `root` respectively.

We have two constructors, one with no arguments (which only sets `root` to null), and another with one argument (the first element to be inserted on to the tree). Then we have a standard `isEmpty()` method to find out if the tree is empty. You'll also notice that implementing a counter for the number of elements inside the tree is not a hard thing to do (very similar to the way we did it in a linked list).

The `getData()` method returns the data of the `root` node. This may not be particularly useful to us right now, but may be needed in some derived class (so, we stick it in there just for convenience). Throughout data structures, and mostly entire programming world, you'll find that certain things are done solely for convenience. Other "convenient" methods are `getLeft()`, `getRight()` and `setData()`.

The two methods we'll be using later (for something useful), are: `insertLeft(pTwoChildNode,Object)`, and `insertRight(pTwoChildNode,Object)`. These provide a nice way to quickly insert something into the left child (sub-tree) of the given node.

The rest are just print methods. The trick about trees are that they can be traversed in many different ways, and these print methods print out the whole tree, in different traversals. All of these are useful, depending on what you're doing, and what type of tree you have. Sometimes, some of them make absolutely no sense, depending on what you're doing.

Printing methods are recursive; a lot of the tree manipulation functions are recursive, since they're described so naturally in recursive structures. A recursive function is a function that calls itself (kind of like `pretrav()`, `intrav()`, and `postrav()` does).

Go over the source, make sure you understand what each function is doing (not a hard task). It's not important for you to understand why we need all these functions at this point (for now, we "just" need them); you'll understand why we need some of them in the next few sections, when we `extend` this class to create a really cool sorting engine.

Comparing Objects...

Comparing Objects in Java can be a daunting task, especially if you have no idea how it's done. In Java, we can only compare variables of `native` type. These include all but the objects (ex: `int`, `float`, `double`, etc.). To compare Objects, we have to make objects with certain properties; properties that will allow us to compare.

We usually create an `interface`, and `implement` it inside the objects we'd like to compare. In our case, we'll call the `interface pComparable`. Interfaces are easy to write, since they're kind of like `abstract` classes.

```
public interface pComparable{
    public int compareTo(Object o);
}
```

As you can see, there is nothing special to simple interfaces. Now, the trick is to `implement` it. You might be saying, why am I covering comparing of objects right in the middle of a binary tree discussion... well, we can't have a binary search tree without being able to compare objects. For example, if we'd like to use integers in our binary search tree, we'll have to design our own integer, and let it have a `pComparable` interface.

Next follows our implementation of `pInteger`, a number with a `pComparable interface`. I couldn't just `extend` the `java.lang.Integer`, since it's `final` (cannot be `extended`) (those geniuses!).

```
public class pInteger implements pComparable{
    int i;

    public pInteger(){
    }
    public pInteger(int j){
        set(j);
    }
    public int get(){
        return i;
    }
    public void set(int j){
        i = j;
    }
    public String toString(){
        return ""+get();
    }
    public int compareTo(Object o){
        if(o instanceof pInteger)
            if(get() > ((pInteger)o).get())
                return 1;
            else if(get() < ((pInteger)o).get())
                return -1;
        return 0;
```

```
        }
}
```

I believe most of the `interface` is self explanatory, except maybe for the `compareTo(Object)` method. In the method, we first make sure that the parameter is of type `pInteger`, and later using casting, and calling methods, we compare the underlying `native` members of `pInteger` and return an appropriate result.

A note on JDK 1.2: In the new versions of the JDK, you won't need to `implement` your own `pComparable`, or your own `pInteger`; since it's built in! There is a `Comparable interface`, and it's already `implemented` by the built in `java.lang.Integer`, `java.lang.String`, and other classes where you might need comparisons. I'm doing it this way only for compatibility with the older versions of JDK. I'll talk about JDK 1.2 features later in this document (hopefully).

Binary Search Trees...

And now, back to the show, or shall I say Binary Trees! A binary tree we'll be designing in this section will be what's known as binary search tree. The reason it's called this is that it can be used to sort numbers (or objects) in a way, that makes it very easy to search them; traverse them. Remember how I've said that traversals only make sense in some specific context, well, in binary search tree, only the in-traversal makes sense; in which numbers (objects) are printed in a sorted fashion. Although I'll show all traversals just for the fun of it.

A binary search tree will `extend` our `pGenericBinaryTree`, and will add on a few methods. One that we definitely need is the `insert()` method; to `insert` objects into a tree with binary search in mind. Well, instead of just talking about it, lets write the source!

```
import pComparable;

public class pBinarySearchTree extends pGenericBinaryTree{

    public pBinarySearchTree(){
        super();
    }

    public pBinarySearchTree(Object o){
        super(o);
    }

    public void print(){
        print(2);
    }

    public void insert(pComparable o){
        pTwoChildNode t,q;
        for(q = null, t = getRoot();
            t != null && o.compareTo(t.getData()) != 0;
            q = t,t = o.compareTo(t.getData()) < 0
                ? t.getLeft():t.getRight());
        if(t != null)
            return;
        else if(q == null)
            setRoot(new pTwoChildNode(o));
        else if(o.compareTo(q.getData()) < 0)
            insertLeft(q,o);
        else
            insertRight(q,o);
    }
}
```

As you can obviously see, the `insert(pComparable)` method is definitely the key to the whole thing. The method starts out by declaring two variables, `'t'`, and `'q'`. It then falls into a `for` loop. The condition inside the `for` loop is that `'t'` does not equal to `null` (since it was initially set to `getRoot()`, which effectively returns the value of `root`), and while the object we're trying to `insert` does not equal to the object already inside the tree.

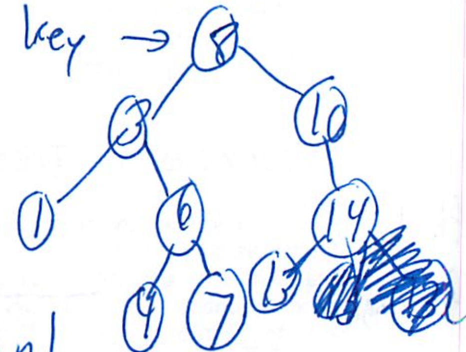
Usually, a binary search tree does not allow duplicate insertions, since they're kind of useless; that's why we're attempting to catch the case where we're trying to `insert` a duplicate. Inside the `for` loop, we set `'q'` to the value of the next node to be examined. We do this by first comparing the data we're inserting with the data in the current node, if it's greater, we set `'t'` to the right node, if less, we set it to the left node (all this is cleverly disguised inside that `for` statement).

We later check the value of `'t'` to make sure we've gotten to the end (or leaf) of the tree. If `'t'` is not `null`, that means we've encountered a duplicate, and we simply `return`. We then check to see if the tree is empty (didn't have a `root`), if it didn't, we create a `new root` by calling `setRoot()` with a newly created node holding the inserted data.

If all else fails, simply `insert` the object into the left or the right child of the leaf node depending on the value of the data. And that's that!

Understanding binary search trees is not easy, but it is the key to some very interesting algorithms. So, if you miss out on the main point here, I suggest you read it again, or get a more formal reference (where I doubt you'll learn more).

Anyway, as it was with our stacks and queues, we always had to test everything, so, lets test it! Below, I give you the test module for the tree.




```
import java.io.*;
import pInteger;
import pBinarySearchTree;

class pBinarySearchTreeTest{
    public static void main(String[] args){
        pBinarySearchTree tree = new pBinarySearchTree();
        pInteger n;
        int i;
        System.out.println("Numbers inserted:");
        for(i=0;i<10;i++){
            tree.insert(n=new pInteger((int)(Math.random()*1000)));
            System.out.print(n+" ");
        }
        System.out.println("\nPre-order:");
        tree.print(1);
        System.out.println("\nIn-order:");
        tree.print();
        System.out.println("\nPost-order:");
        tree.print(3);
    }
}
```

As you can see, it's pretty simple (and similar to our previous tests). It first `inserts` ten `pInteger` (`pComparable`) objects in to the tree, and then traverses the tree in different orders. These different orders print out the whole tree. Since we know it's a binary search tree, the in-order traversal should produce an ordered output. So, lets take a look at the output!

```
Numbers inserted:
500 315 219 359 259 816 304 902 681 334
Pre-order:
500 315 219 259 304 359 334 816 681 902
In-order:
219 259 304 315 334 359 500 681 816 902
Post-order:
304 259 219 334 359 315 681 902 816 500
```

Well, our prediction is confirmed! The in-order traversal did produce sorted results. There is really nothing more I can say about this particular binary search tree, except that it's worth knowing. This is definitely not the fastest (nor was speed an issue), and not necessarily the most useful class, but it sure may proof useful in teaching you how to use trees.

And now, onto something completely different! NOT! We're going to be doing trees for a while... I want to make sure you really understand what they are, and how to use them. (and to show you several tricks other books try to avoid <especially Java books>)

Tree Traversals...

I've talked about tree traversals before in this document, but lets review what I've said. Tree's are created for the sole purpose of traversing them. There are two major traversal algorithms, the depth-first, and breadth-first.

So far, we've only looked at depth-first. Pre-traversal, in-traversal, and post-traversal are subsets of depth-first traversals. The reason it's named depth-first, is because we eventually end up going to the deepest node inside the tree, while still having unseen nodes closer to the root (it's hard to explain, and even harder to understand). Tracing a traversal surely helps; and you can trace that traversal from the previous section (it's only ten numbers!).

The other type of traversal is more intuitive; more "human like." Breadth-first traversal goes through the tree top to bottom, left to right. Lets say you were given a tree to read (sorry, don't have a non-copyrighted picture I can include), you'd surely read it top to bottom, left to right (just like a page of text, or something).

Think of a way you visualize a tree... With the root node on top, and all the rest extending downward. What *Breadth-First* allows us to do is to trace the tree from top to bottom as you see it. It will visit each node at a given tree depth, before moving onto the the next depth.

A lot of the algorithms are centered around *Breadth-First* method. Like the search tree for a *Chess* game. In chess, the tree can be very deep, so, doing a *Depth-First* traversal (search) would be costly, if not impossible. With *Breadth-First* as applied in *Chess*, the program only looks at several moves ahead, without looking too many moves ahead.

The Breadth-First traversal is usually from left to right, but that's usually personal preference. Because the *standard* consul does not allow graphics, the output may be hard to correlate to the actual tree, but I will show how it's done.

As with previous examples, I will provide some modified source that will show you how it's done. An extended `pBinarySearchTree` is shown below:

```
import pTwoChildNode;
import pBinarySearchTree;
import pEasyQueue;

public class pBreadthFirstTraversal extends pBinarySearchTree{

    public void breadth_first(){
        pEasyQueue q = new pEasyQueue();
        pTwoChildNode tmp;
        q.insert(getRoot());
        while(!q.isEmpty()){
            tmp = (pTwoChildNode)q.remove();
            if(tmp.getLeft() != null)
                q.insert(tmp.getLeft());
            if(tmp.getRight() != null)
                q.insert(tmp.getRight());
            System.out.print(tmp.getData()+" ");
        }
    }
}
```

As you can see, the class is pretty simple (only one function). In this demo, we're also using `pEasyQueue`, developed earlier in this document. Since breadth first traversal is not like depth first, we can't use recursion, or stack based methods, we need a queue. Any recursive method can be easily simulated using a stack, not so with breadth first, here, we definitely need a queue.

As you can see, we start by first inserting the `root` node on to the queue, and loop while the queue is not `isEmpty()`. If we have a left node in the node being examined, we insert it in to the queue, etc. (same goes for the right node). Eventually, the nodes inserted in to the queue, get removed, and subsequently, have their left children examined. The process continues until we've traversed the entire tree, from top to bottom, left to right order.

Now, lets test it. The code below is pretty much the same code used to test the tree, with one minor addition; the one to test the breadth-first traversal!

```
import java.io.*;
import pInteger;
import pBinarySearchTree;

class pBreadthFirstTraversalTest{
    public static void main(String[] args){
        pBreadthFirstTraversal tree = new pBreadthFirstTraversal();
        pInteger n;
        int i;
        System.out.println("Numbers inserted:");
        for(i=0;i<10;i++){
            tree.insert(n=new pInteger((int)(Math.random()*1000)));
            System.out.print(n+" ");
        }
        System.out.println("\nPre-order:");
        tree.print(1);
        System.out.println("\nIn-order:");
        tree.print();
        System.out.println("\nPost-order:");
        tree.print(3);
        System.out.println("\nBreadth-First:");
        tree.breadth_first();
    }
}
```

As you can see, nothing too hard. Next, goes the output of the above program, and you and I will have to spend some time on the output.

```
Numbers inserted:
890 474 296 425 433 555 42 286 724 88
Pre-order:
890 474 296 42 286 88 425 433 555 724
In-order:
```

```
42 88 286 296 425 433 474 555 724 890
Post-order:
88 286 42 433 425 296 724 555 474 890
Breadth-First:
890 474 296 555 42 425 724 286 433 88
```

Looking at the output in this format is very abstract and is not very intuitive. Lets just say we have some sort of a tree, containing these numbers above. We were looking at the root node. Now, looking at the output of this program, can you guess what the root node is? Well, it's the first number in breadth-first: `890`. The left child of the root is: `474`. Basically, the tree looks like:

```
             |--[890]
             |
        |--[474]--|
        |         |
   |--[296]--|  [555]--|
   |         |         |
[ 42]--|  [425]--|  [724]
       |         |
   |--[286]     [433]
   |
[ 88]
```

As is evident from this picture, I'm a terrible ASCII artist! (If it's screwed up, sorry).

What you can also see is that if you read the tree form left to right, top to bottom, you end up with the breadth first traversal. Actually, this is a good opportunity for you to go over all traversals, and see how they do it, and if they make sense.

There are many variations to these traversals (and I'll show a few in subsequent sections), for now, however, try to understand these four basic ones.

Well, see you again in the next section, where we examine some ways you can speed up your code and take a look at JDK 1.2 features, which will simplify our life a LOT!

Node Pools...

Since you've looked at node based data structures, and their flexibility, you might wonder why would we want to use anything else. Well, sometimes, the situation comes where the standards are not enough. Where we need that extra boost of performance, which is currently occupied by the Java's garbage collection.

In any environment, allocating and releasing memory is slow! Most of the time, you should try to avoid the procedure, unless you really need it (or it's not in a time critical loop). You certainly wouldn't want to allocate and/or release memory every time you do anything to a list, or a binary tree. What can we do to avoid the natural way? Plenty!

The most common way around these kinds of problems is to use our own allocation technique, which is perfectly suited for our purposes, and will generally be much faster than the current system's approach (but always make sure; system techniques are quite fast as well).

Using our own allocation scheme does imply that we'll have to restrict our selves, but most of the times (when you really need it), the restriction is worth while. What we will be doing is simulating the allocation and release of nodes, using an array of nodes, stored in a linked list fashion. It may seem complex, but in reality, it's not.

This array of nodes, is usually referred to as the *"Node Pool"* (we *"catch"* one when we need one, and *"throw it back"* when we don't). I know, I know, I haven't been the same since that girl on IRC has stolen my humor.

There are two considerations we should think about before we start programming, however. These two are whether we want to make the node-pool itself static or dynamic? Shall we allocate the node-pool with every new instance of the object, or shall we just share one common node-pool for all the objects of that class?

These two approaches are both quite useful in different situations. For example, when was the last time you've ever used more than 52 cards in a deck of cards? That implies that your deck of cards class may have no more than 52 cards at a time; a perfect candidate for a Node Pool! Inside your game, you have this one class, which can only hold 52 cards, no more. Then again, what if your card game has to have several decks? You might want to extend that a little more; like have a separate deck for each deck! (am I making any sense?)

The reason I describe this deck of cards example is a historical one; it was the first program I've ever used with Node Pools (long time ago; in C++). The one static node-pool allows other instances of the class (inside the same program) to know which cards are missing from this universal deck, and which are present. On the other hand, if you have a local (non-static) node-pool, each deck handles it's own cards, and non of the other decks know anything about it.

It is important to understand this concept, since it makes a lot of sense <*sometimes*> (and will allow you to write

robust code that doesn't waste memory). Then again, sometimes it makes absolutely no sense to use a node-pool at all!

The example we will design in the next few sections will deal with a binary tree, which does not use dynamic memory when inserting (or removing) nodes. It will have a non-static node-pool; meaning that you'll have to specify the maximum number of elements when it's instantiated (create an instance of it). I believe the example will be general enough for you to learn to apply node-pools in various different situations. Since I've already showed how to use my own `pComparable` for JDK 1.1 (or lower), this next example will deal only with JDK 1.2 `java.lang.Comparable interface` for most purposes. (A `java.lang.Comparable` interface is worth learning, since it's very useful; I only wonder why they haven't come up with something similar earlier?)

Node Pool Nodes...

As with everything else, there are TONS of ways to write exactly the same thing! With nodes, there is no exception. I prefer to implement node-pool nodes with integer children; where the integer points to the next child inside the node-pool array. This may not be the best *Object Oriented* way to do this, since it's binding the node-pool node to the node-pool tree. However, there are many other solutions to this, and many are more *"elegant"* than the one I'll be using here. So, if you don't like it, implement your own version!

If you've paid close attention to the discussion about regular binary trees (somewhere in this document), you'll find this discussion of node-pool stuff pretty easy and similar to that "regular" stuff. Enough talk, lets go and write this node-pool node!

```
public class pNodePoolTreeNode{

    private Object data;
    private int left;
    private int right;

    public pNodePoolTreeNode(){
        left = right = -1;
        data = null;
    }
    public pNodePoolTreeNode(int l,int r){
        left = l;
        right = r;
        data = null;
    }
    public Object getData(){
        return data;
    }
    public void setData(Object o){
        data = o;
    }
    public int getLeft(){
        return left;
    }
    public void setLeft(int l){
        left = l;
    }
    public int getRight(){
        return right;
    }
    public void setRight(int r){
        right = r;
    }
    public String toString(){
        return new String(data.toString());
    }
}
```

The code above is pretty much self explanatory (if it's not, take a look back at binary tree nodes section, and you'll figure it out). The only significant thing you should notice is that children of the node are now integers, instead of references to other node objects. Although, it later turns out that what these integer children are referencing are in fact nodes.

Well, that was easy enough, are you ready to figure out how a tree actually manipulates these?

Node Pool Generic Trees...

A node-pool tree is not that much different from a regular tree. The only key differences is that it has to maintain a linked list (more or less a "stack") of free nodes, and allocate them when needed.

Another tricky thing is that now, the node's children are pointed to by integers, which are references into a node-pool array; so, watch out. You'll notice that a lot of the "tree stuff" has been re-used from the previous version, and you're right! It makes it a lot easier to understand on your part, and a lot easier to write on mine! (oh, I just ran out of coffee...)

```
import pNodePoolTreeNode;
import java.io.*;

public abstract class pNodePoolArrayTree{

    private int numNodes,nextAvail;
    private pNodePoolTreeNode[] arrayNodes;
    protected int root;

    private void init(){
        int i;
        root = -1;
        arrayNodes = new pNodePoolTreeNode[numNodes];
        nextAvail = 0;
        for(i=0;i<numNodes;i++)
            arrayNodes[i] = new pNodePoolTreeNode(-1,i+1);
        getNode(i-1).setRight(-1);
    }
    public pNodePoolArrayTree(){
        numNodes = 500;
        init();
    }
    public pNodePoolArrayTree(int n){
        numNodes = n;
        init();
    }
    protected int getNode(){
        int i = nextAvail;
        nextAvail = getNode(nextAvail).getRight();
        if(nextAvail == -1){
            nextAvail = i;
            return -1;
        }
        return i;
    }
    protected void freeNode(int n){
        getNode(n).setRight(nextAvail);
        nextAvail = n;
    }
    public boolean isEmpty(){
        return getRoot() == -1;
    }
    public boolean isFull(){
        int i = getNode();
        if(i != -1){
            freeNode(i);
            return false;
        }
        return true;
    }
    protected Object getData(){
        if(isEmpty())
            return null;
        return getNode(getRoot()).getData();
    }
    protected void setData(Object o){
        if(isEmpty())
            root = getNode();
        getNode(root).setData(o);
        getNode(root).setLeft(-1);
        getNode(root).setRight(-1);
    }
    protected int getLeft(){
        if(isEmpty())
            return -1;
        return getNode(getRoot()).getLeft();
    }
    protected void setLeft(int n){
        if(isFull())
            return;
        if(isEmpty()){
            root = getNode();
            getNode(getRoot()).setRight(-1);
```

```
        }
        getNode(getRoot()).setLeft(n);
    }
    protected void setRight(int n){
        if(isFull())
            return;
        if(isEmpty()){
            root = getNode();
            getNode(getRoot()).setLeft(-1);
        }
        getNode(getRoot()).setRight(n);
    }
    protected int getRight(){
        if(isEmpty())
            return -1;
        return getNode(root).getRight();
    }
    protected int getRoot(){
        return root;
    }
    protected pNodePoolTreeNode getNode(int n){
        if(n != -1)
            return arrayNodes[n];
        else return null;
    }
    protected void insertLeft(int node,Object o){
        if((node != -1) && (getNode(node).getLeft() == -1)
            && !isFull()){
            int i = getNode();
            getNode(i).setData(o);
            getNode(i).setRight(-1);
            getNode(node).setLeft(i);
        }
    }
    protected void insertRight(int node,Object o){
        if((node != -1) && (getNode(node).getRight() == -1)
            && !isFull()){
            int i = getNode();
            getNode(i).setData(o);
            getNode(i).setRight(-1);
            getNode(node).setRight(i);
        }
    }
    public void print(int mode){
        switch(mode){
        case 1:
            pretrav();
            break;
        case 2:
            intrav();
            break;
        case 3:
            postrav();
            break;
        }
    }
    public void pretrav(){
        pretrav(getRoot());
    }
    private void pretrav(int p){
        if(p == -1)
            return;
        System.out.print(getNode(p).getData()+" ");
        pretrav(getNode(p).getLeft());
        pretrav(getNode(p).getRight());
    }
    public void intrav(){
        intrav(getRoot());
    }
    private void intrav(int p){
        if(p == -1)
            return;
        intrav(getNode(p).getLeft());
        System.out.print(getNode(p).getData()+" ");
        intrav(getNode(p).getRight());
    }
    public void postrav(){
        postrav(getRoot());
    }
```

```
    private void postrav(int p){
        if(p == -1)
            return;
        postrav(getNode(p).getLeft());
        postrav(getNode(p).getRight());
        System.out.print(getNode(p).getData()+" ");
    }
}
```

There are not that many hard methods in this one; so, lets focus in on the hard ones, and I'll leave you to figure out the *"easy"* ones yourself.

One thing I'd like to mention of the top so that it causes as little confusion as possible: now that we're not using objects, but integers, we still have to note the "`null` node" somehow, so, in our example, I'm using `-1` as the "`null` node" indicator.

Data members `numNodes`, `nextAvail`, and `arrayNodes` are there to keep track of the linked list of free and occupied nodes. It's not exactly a linked list (although many people like to think of it that way), it's more or less a simple array stack we've implemented at the beginning of this document. The `numNodes` is the maximum number of nodes inside this tree; we could just as well substitute it for `arrayNodes.length`, but I think this "separation" is more convenient.

The `nextAvail` member is the one that points to the next available node. Our allocation methods, `getNode()` and `freeNode(int n)`, uses this particular *pointer* to allocate and release nodes. And `arrayNodes` is just an array holding the node-pool. All of the integer child references are into this `arrayNodes` array; which we allocate dynamically during instantiation (creation) of the object.

Inside the `init()` function, we allocate the `arrayNodes` array to hold the node-pool. We later loop through all the nodes inside of it, and turn them into a linked list (with right child pointing to the next available node and left child being `-1`). The last node in the list is apparently marked `-1`; since it's last!

I suggest you go over the source for `getNode()` and `freeNode(int n)`; try to understand what it's doing. Which is nothing more complex than what we've already done! (look over the linked list description if you find it confusing)

The rest of the functions are pretty much self explanatory (you should be able to figure them out). Most of them are just a port of the old tree functions to use integer children with node-pool nodes. Well, are you ready for a binary sort tree implementing a node-pool (ready or not, we're going there)!

Node Pool Sort Trees...

As I've mentioned before, node-pool implementations are fairly similar to the regular ones, so, our old sort tree will not change much. The only things that will change however, are the node handling methods. Since we already know what needs to be done (review the binary sort example described previously in this document), we can just go ahead and write the source.

```
import pNodePoolTreeNode;
import pNodePoolArrayTree;
import java.lang.*;

public class pNodePoolSortArrayTree extends pNodePoolArrayTree{

    public pNodePoolSortArrayTree(){
        super();
    }
    public pNodePoolSortArrayTree(int n){
        super(n);
    }
    public void print(){
        print(2);
    }
    public void insert(Comparable obj){
        int t,q = -1;
        t = getRoot();
        while(t != -1 && !(obj.equals(getNode(t).getData()))){
            q = t;
            if(obj.compareTo(getNode(t).getData()) < 0)
                t = getNode(t).getLeft();
            else
                t = getNode(t).getRight();
        }
        if(t != -1)
            return;
        if(q == -1){
            setData(obj);
```

```
            return;
        }
        if(obj.compareTo(getNode(q).getData()) < 0)
            insertLeft(q,obj);
        else
            insertRight(q,obj);
    }
}
```

Now, the following only makes sense inside a JDK 1.2 (or above) context. Earlier version of the JDK do not support the `Comparable interface`. I just though it would be nice to show how to use all these cool interfaces introduced with JDK 1.2!

The code does nothing special, and chances are, you can figure it out on your own. The only seeming difference between this source and the one presented earlier, is that in this one, whenever we test for a null node, we're not testing the node, we're testing whether if it has value of `-1`. Other than that, the code looks almost identical.

This code is pretty bad, however, in illustrating the beauty of Object Oriented programming. With our approach, we needed to change the whole thing just to implement node-pools. Ideally, we should strive to only have to change one part of the system. For example, we could have only changed the `abstract` parent class, and leave everything else as it was. Or we could have changed the node a little, and make the node itself support stuff, but I just thought this was a clearer approach, without dragging all that Object Oriented stuff into the issue.

I think I've made this Node Pool section longer than it should be; so, let me quickly wrap it up by getting right to the point. Lets write a test driver (which will be an almost identical copy of the old test driver).

```
import java.io.*;
import pNodePoolSortArrayTree;

public class pNodePoolSortArrayTreeTest{

    public static void main(String[] args){
        pNodePoolSortArrayTree tree =
            new pNodePoolSortArrayTree(100);
        int i,n;
        System.out.println("Numbers inserted:");
        for(i=0;i<10;i++){
            tree.insert(new Integer(n=(int)(Math.random()*1000)));
            System.out.print(n+" ");
        }
        System.out.println("\nPre-order:");
        tree.print(1);
        System.out.println("\nIn-order:");
        tree.print(2);
        System.out.println("\nPost-order:");
        tree.print(3);
    }
}
```

You'll notice that the code above uses `java.lang.Integer` object, instead of our own `pInteger` one. That's because in JDK 1.2, an `Integer` is `implementing Comparable interface`, simplifying a whole lot of stuff. Other than that, the code works identically to the one shown earlier in the document. Just to be complete, however, lets include the output from the above program.

```
Numbers inserted:
103 82 165 295 397 828 847 545 766 384
Pre-order:
103 82 165 295 397 384 828 545 766 847
In-order:
82 103 165 295 384 397 545 766 828 847
Post-order:
82 165 295 397 384 828 545 766 847 103
```

I guess that wraps it up. (finally!) Now, I'm gonna take a little break, and try to come up with some cool programs that use these data structures. The only way to actually learn what they are, and how to use them, is to see them in action. And some of these can be VERY useful in a wide variety of applications, ranging form databases, to graphics programming.

So, now that you know all the basics of everything about data structures, the remainder of this document will mostly concentrate on interesting approaches, and algorithms using these. (You'll be surprised how far a binary sort tree can go when it comes to creating virtual 3D worlds!)

Priority Vectors, etc.

We've learned about the structure of vectors, lists, queues, etc., but what if you need some order in the way data is arranged? You have several options to this; you can simply implement a sort method inside the data storage class, or you can make the class itself a priority one.

Priority classes are widely used in programs because they simplify a lot of things. For example, you don't have to worry about sorting data, the class itself does it for you. Imagine a simulation of a real world queue at a doctor's office, where people with life threatening injuries get priority over everyone else. The queue has to accept all people, and sort them according to the seriousness of their illness; with most serious ending up seeing the doctor first.

The priority concept can be applied anywhere, not just to queues. It can easily be extended to lists, and vectors. In this section, we will talk about creating a priority vector (since it's fairly simple). We will also cover some *"well known"* sorting algorithms.

How simple is it? The answer is: very! All we need is to extend the `java.util.Vector` class, add one insert function, and we are done. And here's the source:

```
import java.lang.Comparable;

public class pPriorityVector extends java.util.Vector{
    public void pAddElement(Comparable o){
        int i,j = size();
        for(i=0;i<j&&(((Comparable)(elementAt(i))).compareTo(o)<0);i++);
        insertElementAt(o,i);
    }
}
```

As you can see, it's VERY simple. We simply use this class the way we would use any other `java.util.Vector`, accept, when we use `pAddElement(Comparable)` method, we are inserting in accenting order. We could have just as well written a descending order one; I think you get the picture...

Lets test this class, and see if it really works.

```
import java.io.*;
import java.util.*;
import pPriorityVector;

class pPriorityVectorTest{
    public static void main(String[] args){
        pPriorityVector v = new pPriorityVector();
        System.out.print("starting...\nadding:");
        for(int i=0;i<10;i++){
            Integer j = new Integer((int)(Math.random()*100));
            v.pAddElement(j);
            System.out.print(" " + j);
        }
        System.out.print("\nprinting:");
        Enumeration enum = v.elements();
        while(enum.hasMoreElements())
            System.out.print(" "+(Integer)enum.nextElement());
        System.out.println("\nDone ;-)");
    }
}
```

The above test program simply inserts ten random `java.lang.Integer` objects, and then prints them out. If our class works, the output should produce a sorted list of number. Our prediction is correct, the output follows.

```
starting...
adding: 95 85 62 16 39 73 84 43 77 61
printing: 16 39 43 61 62 73 77 84 85 95
Done ;-)
```

As you can see, we are inserting numbers in an unsorted order, and get a sorted list when we finally print them out. The technique illustrated above sorts the numbers upon insertion, some algorithms do sorting when retrieving data.

Sorting...

Sorting, in general, means arranging something in some meaningful manner. There are TONS of ways of sorting things, and we will only look at the most popular ones. The one we won't cover here (but one you should know) is Quick Sort. You can find a really good Quick Sort example in the JDK's sample applets. We have also learned that we can sort using binary trees, thus, in this section, we won't cover that approach.

We will be using JDK 1.2's `java.lang.Comparable` interface in this section, thus, the examples will not work with earlier versions of the JDK. The sorting algorithms we'll look at in this section are: *Bubble Sort*, *Selection Sort*, *Insertion*

Sort, *Heap Sort*, and lastly, *Merge Sort*. If you think that's not enough, you're welcome to find other algorithms from other sources. One source I'd recommend is the JDK's sources. The JDK has lots of code, and lots of algorithms that may well be very useful.

We won't exactly cover the algorithms; I will simply illustrate them, and you'll have to figure out how they work from the source. (By tracing them.) I don't have the time to go over each and every algorithm; I'm just taking my old C source, converting it to Java, and inserting it into this document ;-)

```
import java.io.*;
import java.util.*;
import java.lang.*;

public class pGeneralSorting{

    public static void BubbleSort(Comparable[] a){
        boolean switched = true;
        for(int i=0;i<a.length-1 && switched;i++){
            switched = false;
            for(int j=0;j<a.length-i-1;j++)
                if(a[j].compareTo(a[j+1]) > 0){
                    switched = true;
                    Comparable hold = a[j];
                    a[j] = a[j+1];
                    a[j+1] = hold;
                }
        }
    }

    public static void SelectionSort(Comparable[] a){
        for(int i = a.length-1;i>0;i--){
            Comparable large = a[0];
            int indx = 0;
            for(int j = 1;j <= i;j++)
                if(a[j].compareTo(large) > 0){
                    large = a[j];
                    indx = j;
                }
            a[indx] = a[i];
            a[i] = large;
        }
    }

    public static void InsertionSort(Comparable[] a){
        int i,j;
        Comparable e;
        for(i=1;i<a.length;i++){
            e = a[i];
            for(j=i-1;j>=0 && a[j].compareTo(e) > 0;j--)
                a[j+1] = a[j];
            a[j+1] = e;
        }
    }

    public static void HeapSort(Comparable[] a){
        int i,f,s;
        for(i=1;i<a.length;i++){
            Comparable e = a[i];
            s = i;
            f = (s-1)/2;
            while(s > 0 && a[f].compareTo(e) < 0){
                a[s] = a[f];
                s = f;
                f = (s-1)/2;
            }
            a[s] = e;
        }
        for(i=a.length-1;i>0;i--){
            Comparable value = a[i];
            a[i] = a[0];
            f = 0;
            if(i == 1)
                s = -1;
            else
                s = 1;
            if(i > 2 && a[2].compareTo(a[1]) > 0)
                s = 2;
            while(s >= 0 && value.compareTo(a[s]) < 0){
                a[f] = a[s];
```

```
                f = s;
                s = 2*f+1;
                if(s+1 <= i-1 && a[s].compareTo(a[s+1]) < 0)
                    s = s+1;
                if(s > i-1)
                    s = -1;
            }
            a[f] = value;
        }
    }

    public static void MergeSort(Comparable[] a){
        Comparable[] aux = new Comparable[a.length];
        int i,j,k,l1,l2,size,u1,u2;
        size = 1;
        while(size < a.length){
            l1 = k = 0;
            while((l1 + size) < a.length){
                l2 = l1 + size;
                u1 = l2 - 1;
                u2 = (l2+size-1 < a.length) ?
                    l2 + size-1:a.length-1;
                for(i=l1,j=l2;i <= u1 && j <= u2;k++)
                    if(a[i].compareTo(a[j]) <= 0)
                        aux[k] = a[i++];
                    else
                        aux[k] = a[j++];
                for(;i <= u1;k++)
                    aux[k] = a[i++];
                for(;j <= u2;k++)
                    aux[k] = a[j++];
                l1 = u2 + 1;
            }
            for(i=l1;k < a.length;i++)
                aux[k++] = a[i];
            for(i=0;i < a.length;i++)
                a[i] = aux[i];
            size *= 2;
        }
    }

    public static void main(String[] args){
        Integer[] a = new Integer[10];
        System.out.print("starting...\nadding:");
        for(int i=0;i<a.length;i++){
            a[i] = new Integer((int)(Math.random()*100));
            System.out.print(" " + a[i]);
        }
        BubbleSort(a);
        System.out.print("\nprinting:");
        for(int i=0;i<a.length;i++){
            System.out.print(" " + a[i]);
        }
        System.out.println("\nDone ;-)");
    }
}
```

The above program both illustrates the algorithms, and tests them. Some of these may look fairly complicated; usually, the more complicated a sorting algorithm is, the faster and/or more efficient it is. That's the reason I'm not covering Quick Sort; I'm too lazy to convert that huge thing! Some sample output from the above program follows:

```
starting...
adding: 22 63 33 19 82 59 70 58 98 74
printing: 19 22 33 58 59 63 70 74 82 98
Done ;-)
```

I think you get the idea bout sorting. You can always optimize the above sorting methods, use native types, etc. You can also use these in derived classes from `java.util.Vector`, using the `java.util.Vector` data directly. And just when you though we were done with sorting, here we go again...

Sorting JDK 1.2 Style...

Wasn't the last section scary and a bit confusing? If you said *"yes,"* then you're not alone. Java implementers also think so; that's why a lot of the sorting thingies are already built in! JDK 1.2 introduced the `Collection` interface; using it, we can do all kinds of data structures effects like sorting and searching. For example, to sort a `Vector`, all we need to

do is call a `sort()` method of the `java.util.Collections` class.

```
import java.io.*;
import java.util.*;

public class pJDKVectorSortingUp{
    public static void main(String[] args){
        Vector v = new Vector();
        System.out.print("starting...\nadding:");
        for(int i=0;i<10;i++){
            Integer j = new Integer((int)(Math.random()*100));
            v.addElement(j);
            System.out.print(" " + j);
        }
        Collections.sort(v);
        System.out.print("\nprinting:");
        Enumeration enum = v.elements();
        while(enum.hasMoreElements())
            System.out.print(" "+(Integer)enum.nextElement());
        System.out.println("\nDone ;-)");
    }
}
```

See how simple it is? The output follows...

```
starting...
adding: 91 37 16 53 11 15 89 44 90 58
printing: 11 15 16 37 44 53 58 89 90 91
Done ;-)
```

All this is nice and useful, but what if you need descending order, instead of the *"default"* accenting one? Guess what?, the implementers of the language have though about that as well! We have a `java.util.Comparator` interface. With this `Comparator`, we can specify our own compare function, making it possible to sort in descending order (or any other way we want... i.e.: sorting absolute values, etc.).

So for objects - how do ya want to sort?

```
import java.io.*;
import java.util.*;

public class pJDKVectorSortingDown implements Comparator{

    public int compare(Object o1,Object o2){
        return -((Comparable)o1).compareTo(o2);
    }

    public static void main(String[] args){
        Vector v = new Vector();
        System.out.print("starting...\nadding:");
        for(int i=0;i<10;i++){
            Integer j = new Integer((int)(Math.random()*100));
            v.addElement(j);
            System.out.print(" " + j);
        }
        Collections.sort(v,new pJDKVectorSortingDown());
        System.out.print("\nprinting:");
        Enumeration enum = v.elements();
        while(enum.hasMoreElements())
            System.out.print(" "+(Integer)enum.nextElement());
        System.out.println("\nDone ;-)");
    }
}
```

As you can see, this time, we're sending a `Comparator` object to the `Collections.sort()` method. This technique is really cool and useful. The output from the above program follows:

```
starting...
adding: 9 96 58 64 13 99 91 55 51 95
printing: 99 96 95 91 64 58 55 51 13 9
Done ;-)
```

I suggest you go over all those JDK classes and their methods. There is a LOT of useful stuff there. Now, say good bye to sorting for a while; we're moving into something totally different; NOT!

Sorting using Quicksort...

Sorting using Quicksort is something that I wasn't planning to cover in this document due to the fact that the

algorithm solely depends on the data it receives. If the data has certain properties, quicksort is one of the fastest, if not, quicksort can be excruciatingly slow. By now, you probably already figured out that *bubble sort* (*covered earlier*) is pretty slow; put it another way: *forget about using bubble sort for anything important*.

Bubble sort tends to be `O(n^2)`; in other words, pretty slow. Now, quicksort can perform quite fast, on average about `O(n log n)`, but it's worst case, is a humiliating `O(n^2)`. For quicksort, the worst case is usually when the data is already sorted. There are many algorithms to make this *"less likely to occur,"* but it does happen. (the `O notation` is paraphrased *"on the order of"*)

If you need a *no-worst-case* fast sorting algorithm, then by all means, use *heap sort* (*covered earlier*). In my opinion, heap sort is more elegant than any other sort <it is *in place* `O(n log n)` *sort*>. However, on average, it does tend to perform twice as slow as quicksort.

UPDATE: I've recently attended a conference at the [insert fancy name here] Science Society, and heard a nice talk by a professor from Princeton. The talk was totally dedicated to Quicksort. Apparently, if you modify the logic a bit, you can make Quicksort optimal! This is a very grand discovery! (unfortunately, I didn't write down the name of anybody; not even the place!) The idea (among other things) involves moving duplicate elements to one end of the list, and at the end of the run, move all of them to the middle, then sort the left and right sides of the list. Since now Quicksort performs well with lots of duplicate values (really well actually ;-), you can create a radix-like sort that uses embedded Quicksort to quickly (very quickly) sort things. There's also been a Dr.Dobbs article (written by that same professor ;-) on this same idea, and I'm sorry for not having any more references than I should. I still think this is something that deserved to be mentioned.

We will start out by writing a general (*simplest*) implementation of quicksort. After we thoroughly understand the algorithm we will re-write it implementing all kinds of tricks to make it faster.

Quicksort is naturally recursive. We partition the array into two sub-arrays, and then re-start the algorithm on each of these sub-arrays. The partition procedure involves choosing some object (usually, already in the array); If some other object is greater than the chosen object, it is added to one of the sub-arrays, if it's less than the chosen object, it's added to another sub-array. Thus, the entire array is partitioned into two sub-arrays, with one sub-array having everything that's larger than the chosen object, and the other sub-array having everything that's smaller.

The algorithm is fairly simple, and it's not hard to implement. The tough part is making it fast. The implementation that follows is recursive and is not optimized, which makes this function inherently slow (most sorting functions try to avoid recursion and try to be as fast as possible). If you need raw speed, you should consider writing a native version of this algorithm.

```
public class pSimpleQuicksort{

    public static void qsort(Comparable[] c,int start,int end){
        if(end <= start) return;
        Comparable comp = c[start];
        int i = start,j = end + 1;
        for(;;){
            do i++; while(i<end && c[i].compareTo(comp)<0);
            do j--; while(j>start && c[j].compareTo(comp)>0);
            if(j <= i)   break;
            Comparable tmp = c[i];
            c[i] = c[j];
            c[j] = tmp;
        }
        c[start] = c[j];
        c[j] = comp;
        qsort(c,start,j-1);
        qsort(c,j+1,end);
    }

    public static void qsort(Comparable[] c){
        qsort(c,0,c.length-1);
    }

    public static void main(String[] args){
        int i;
        Integer[] arr = new Integer[20];
        System.out.println("inserting: ");
        for(i=0;i<arr.length;i++){
            arr[i] = new Integer((int)(Math.random()*99));
            System.out.print(arr[i]+" ");
        }
        pSimpleQuicksort.qsort(arr);
        System.out.println("\nsorted: ");
        for(i=0;i<arr.length;i++)
            System.out.print(arr[i]+" ");
        System.out.println("\nDone ;-)");
    }
```

```
}
```

As you can see, the `qsort()` functions itself is fairly short. One of them is just a wrapper for the other one. The idea is that `qsort()` receives the array, and then the `start`, and `end` pointer between which you want everything sorted. So, the starting call starts at array position `zero`, and ends at the last valid position in the array. Some sample output follows:

```
inserting:
58 52 82 27 23 67 37 63 68 18 95 41 87 6 53 85 65 30 10 3
sorted:
3 6 10 18 23 27 30 37 41 52 53 58 63 65 67 68 82 85 87 95
Done ;-)
```

The sorts starts out by first checking to see if the `end` pointer is less then or equal to the `start` pointer. It if is less, then there is nothing to sort, and we return. If they are equal, we have only one element to sort, and an element by itself is always already sorted, so we return.

If we did not return, we make a pick. In our example, we simply choose the first element in the array to use as our partition element (some call it *pivot* element). To some people, this is the most critical point of the sort, since depending on what element you choose, you'll get either good performance, or bad. A lot of the times, instead of choosing the first, people choose the last, or take a median of the first, last and middle elements. Some even have a random number generator to pick a random element. However, all these techniques are useless against random data, in which case, all those tricky approaches can actually prove to worsen results. Then again, most of the times, they do work quite nicely... (it really depends on the type of data you are working with)

After we have picked the element, we setup `i` and `j`, and fall into the a `for` loop. Inside that loop, we have two inner loops. Inside the first inner loop, we scan the array looking for the first element which is larger than our picked element, moving from left to right of the array. Inside the second inner loop, we scan to find an element smaller than the picked, moving from right to left in the array. Once we've found them (fallen out of both loops), we check to see if the pointers haven't crossed (since if they did, we are done). We then swap the elements, and continue on to the next iteration of the loop.

You should notice that in all these loops, we are doing one simple thing. We are making sure that only one element gets to it's correct position. We deal with one element at a time. After we find that correct position of our chosen element, all we need to do is sort the elements on it's right, and left sides, and we're done. You should also notice that the algorithm above is lacking in optimization. The inner loops, where most of the time takes place are not as optimized as they should be. However, for the time being, try to understand the algorithm; and we will get back to optimizing a bit later.

When we fall out of the outer loop, we put our chosen element into it's correct position, calculate the upper and lower bounds of the left and right arrays (from that chosen elements), and recursively call the method on these new computed arrays.

That's basically it for the *general* algorithm. In the next section, you'll be amazed at how much faster we can make this work. Basically, in the next section, we will implement a sort function to use everywhere (it will be faster than most other approaches).

Optimizing Quicksort...

Optimizing Quicksort written in Java is not such a great task. Whatever tricks we use, we will still be restrained by the speed of the Java Virtual Machine (JVM). In this section, we will talk about algorithmic improvements, rather than quirky tricks.

The first thing what we should do is look at the above code of the un-optimized sort, and see what can be improved. One thing that should be obvious is that it's way too much work if the array is very small. There are a LOT simpler sorts available for small arrays. For example, simple *insertion sort* has almost no overhead, and on small arrays, is actually faster than *quicksort*! This can be fixed rather easily, by including an `if` statement to see if the size of the array is smaller than some particular value, and doing a simple *insertion sort* if it is. This threshold value can only be determined from doing actual experiments with sample data. However, usually any value less than `15` is pretty good; and that is why we will choose `7` for our upcoming example.

What makes quicksort so fast is the simplicity of it's inner loops. In our previous example, we were doing extra work in those inner loops! We were checking for the array bounds; we should have made some swaps beforehand to make sure that the inner loops are as simple as possible. Basically, the idea is to make sure that the first element in the array is less than or equal to the second, and that the second is less than or equal to the last. Knowing that, we can remove those array bounds checking from the inner loops (in some cases speeding up the algorithm by about two times).

Another thing that should be obvious is recursion. Because sort methods are usually very performance hungry, we would like to remove as much function calls as possible. This includes getting rid of recursion. The way we can get rid of recursion is using a stack to store the upper and lower bounds of arrays to be sorted, and using a loop to control the sort.

A question automatically arises: *How big should that stack be?* A simple answer is: *It should be big enough. (you'll be surprised how many books avoid this topic!)* We cannot just use `java.util.Vector` object for it's dynamic growing ability, since that would be way too much overhead for simple stack operations that should be as quick as possible. Now, what would be a *big enough* stack? Well, a formula below calculates the size:

```
size = 2 * ln N / ln 2
```

Where `ln` is natural logarithm, and `N` is the number of elements in the array. Which all translates to:

```
size = (int)(Math.ceil(2.0*Math.log(array.length)/Math.log(2.0));
```

Believe it or not, but it's actually not worth using this equation. If you think about it, you will notice that the size of the stack grows VERY slowly. For example, if the `array.length` is `0xFFFFFFFF` in length (highest `32 bit` integer value), then size will be `64`! We will make it `128` just in case we will ever need it for `64 bit` integers. (If you don't like magic values inside your program, then by all means, use the equation.)

Having gotten to this point, we are almost ready to implement our optimized version of quicksort. I say *almost* because it is still not optimized to it's fullest. If we were using native types instead of `Comparable` objects, then the whole thing would be faster. If we implementing it as native code, it would be even faster. Basically, most other speed-ups are left up to these system or program specific quirky optimizations. And now, here's our optimized version:

```
import java.lang.*;
import java.io.*;

public class pQuicksort{

    public static void sort(Comparable[] c){
        int i,j,left = 0,right = c.length - 1,stack_pointer = -1;
        int[] stack = new int[128];
        Comparable swap,temp;
        while(true){
            if(right - left <= 7){
                for(j=left+1;j<=right;j++){
                    swap = c[j];
                    i = j-1;
                    while(i>=left && c[i].compareTo(swap) > 0)
                        c[i+1] = c[i--];
                    c[i+1] = swap;
                }
                if(stack_pointer == -1)
                    break;
                right = stack[stack_pointer--];
                left = stack[stack_pointer--];
            }else{
                int median = (left + right) >> 1;
                i = left + 1;
                j = right;
                swap = c[median]; c[median] = c[i]; c[i] = swap;
                /* make sure: c[left] <= c[left+1] <= c[right] */
                if(c[left].compareTo(c[right]) > 0){
                    swap = c[left]; c[left] = c[right]; c[right] = swap;
                }if(c[i].compareTo(c[right]) > 0){
                    swap = c[i]; c[i] = c[right]; c[right] = swap;
                }if(c[left].compareTo(c[i]) > 0){
                    swap = c[left]; c[left] = c[i]; c[i] = swap;
                }
                temp = c[i];
                while(true){
                    do i++; while(c[i].compareTo(temp) < 0);
                    do j--; while(c[j].compareTo(temp) > 0);
                    if(j < i)
                        break;
                    swap = c[i]; c[i] = c[j]; c[j] = swap;
                }
                c[left + 1] = c[j];
                c[j] = temp;
                if(right-i+1 >= j-left){
                    stack[++stack_pointer] = i;
                    stack[++stack_pointer] = right;
                    right = j-1;
                }else{
                    stack[++stack_pointer] = left;
                    stack[++stack_pointer] = j-1;
                    left = i;
                }
            }
        }
```

```
    }

    public static void main(String[] args){
        int i;
        Integer[] arr = new Integer[20];
        System.out.println("inserting: ");
        for(i=0;i<arr.length;i++){
            arr[i] = new Integer((int)(Math.random()*99));
            System.out.print(arr[i]+" ");
        }
        pQuicksort.sort(arr);
        System.out.println("\nsorted: ");
        for(i=0;i<arr.length;i++)
            System.out.print(arr[i]+" ");
        System.out.println("\nDone ;-)");
    }
}
```

The output closely follows:

```
inserting:
17 52 88 79 91 41 31 57 0 29 87 66 94 22 19 30 76 85 61 16
sorted:
0 16 17 19 22 29 30 31 41 52 57 61 66 76 79 85 87 88 91 94
Done ;-)
```

This new sort function is a bit larger than most other sorts. It starts out by setting `left` and `right` pointers, and the `stack`. Stack allocation could be moved outside the function, but making it static is not a good choice since that introduces all kinds of multithreading problems.

We then fall into an infinite loop in which we have an `if()` and an `else` statement. The `if()` statement finds out if we should do a simple *insertion sort*, `else`, it will do quicksort. I will not explain *insertion sort* here (since you should already be familiar with it). So, after *insertion sort*, we see if the `stack` is not empty, if it is, we `break` out of the infinite loop, and `return` from the function. If we don't `return`, we pop the `stack`, set new `left` and `right` pointers (from the stack), and continue on with the next iteration of the loop.

Now, if threshold value passed, and we ended up doing quicksort we first find the `median`. This `median` is used here to make the case with bad performance *less likely to occur*. It is useless against random data, but does help if the data is in somewhat sorted order.

We swap this `median` with the second value in the array, and make sure that the first value is less than or equal than the second, and the second is less than or equal than the last. After that, we pick our partition element (or pivot), and fall into an infinite loop of finding that pivot's correct place.

Notice that the most inner loops are fairly simple. Only one increment or decrement operation, and a compare. The compare could be improved quite a bit by using native types; instead of calling a function. The rest of this part of the sort is almost exactly like in the un-optimized version.

After we find the correct position of the pivot variable, we are ready to continue the sort on the right sub-array, and on the left sub-array. What we do next is check to see which one of these sub-arrays is bigger. We then insert the bigger sub-array bounds on to the stack, and setup new `left` and `right` pointers so that the smaller sub-array gets processed next.

I guess that's it for quicksort. If you still don't understand it, I doubt any other reference would be of any help. Basically go through the algorithm; tracing it a few times, and eventually, it will seem like second nature to you. The above function is good enough for most purposes. I've sorted HUGE arrays (*~100k*) of random data, and it performed quite well.

Radix Sort...

Radix sort is one of the nastiest sorts that I know. This sort can be quite fast when used in appropriate context, however, to me, it seems that the context is never appropriate for radix sort.

The idea behind the sort is that we sort numbers according to their base (sort of). For example, lets say we had a number 1024, and we break it down into it's basic components. The 1 is in the thousands, the 0 is in the hundreds, the 2 is in the tens, and 4 is in some units. Anyway, given two numbers, we can sort them according to these bases (i.e.: 100 is greater than 10 because first one has more 'hundreds').

In our example, we will start by sorting numbers according to their least significant bits, and then move onto more significant ones, until we reach the end (at which point, the array will be sorted). This can work the other way as well, and in some cases, it's even more preferable to do it 'backwards'.

Sort consists of several passes though the data, with each pass, making it more and more sorted. In our example, we won't be overly concerned with the actual decimal digits; we will be using base 256! The workings of the sort are shown below:

Numbers to sort: `23 45 21 56 94 75 52` we create `ten` queues (one queue for each digit): `queue[0 .. 9]`

We start going thought the passes of sorting it, starting with least significant digits.

```
queue[0] = { }
queue[1] = {21}
queue[2] = {52}
queue[3] = {23}
queue[4] = {94}
queue[5] = {45,75}
queue[6] = {56}
queue[7] = { }
queue[8] = { }
queue[9] = { }
```

Notice that the queue number corresponds to the least significant digit (i.e.: queue `1` holding `21`, and queue `6` holding `56`). We copy this queue into our array (top to bottom, left to right) Now, numbers to be sorted: `21 52 23 94 45 75 56` We now continue with another pass:

```
queue[0] = { }
queue[1] = { }
queue[2] = {21,23}
queue[3] = { }
queue[4] = {45}
queue[5] = {52,56}
queue[6] = { }
queue[7] = {75}
queue[8] = { }
queue[9] = {94}
```

Notice that the queue number corresponds to the most significant digit (i.e.: queue `4` holding `45`, and queue `7` holding `75`). We copy this queue into our array (top to bottom, left to right) and the numbers are sorted: `21 23 45 52 56 75 94`

Hmm... Isn't that interesting? Anyway, you're probably wondering how it will all work within a program (or more precisely, how much book keeping we will have to do to make it work). Well, we won't be working with `10` queues in our little program, we'll be working with `256` queues! We won't just have least significant and most significant bits, we'll have a whole range (from `0xFF` to `0xFF000000`).

Now, using arrays to represent Queues is definitely out of the question (most of the times) since that would be wasting tons of space (think about it if it's not obvious). Using dynamic allocation is also out of the question, since that would be extremely slow (since we will be releasing and allocating nodes throughout the entire sort). This leaves us with little choice but to use node pools. The node pools we will be working with will be really slim, without much code to them. All we need are nodes with two integers (one for the data, the other for the link to next node). We will represent the entire node pool as a two dimensional array, where the height of the array is the number of elements to sort, and the width is two.

Anyway, enough talk, here's the program:

Since we will be instanciating a lot of them

```
import java.lang.*;
import java.io.*;

public class RadixSort{

    public static void radixSort(int[] arr){
        if(arr.length == 0)
            return;
        int[][] np = new int[arr.length][2];
        int[] q = new int[0x100];
        int i,j,k,l,f = 0;
        for(k=0;k<4;k++){
            for(i=0;i<(np.length-1);i++)
                np[i][1] = i+1;
            np[i][1] = -1;
            for(i=0;i<q.length;i++)
                q[i] = -1;
            for(f=i=0;i<arr.length;i++){
                j = ((0xFF<<(k<<3))&arr[i])>>(k<<3);
                if(q[j] == -1)
                    l = q[j] = f;
                else{
                    l = q[j];
                    while(np[l][1] != -1)
```

```
                        l = np[l][1];
                    np[l][1] = f;
                    l = np[l][1];
                }
                f = np[f][1];
                np[l][0] = arr[i];
                np[l][1] = -1;
            }
            for(l=q[i=j=0];i<0x100;i++)
                for(l=q[i];l!=-1;l=np[l][1])
                        arr[j++] = np[l][0];
        }
    }

    public static void main(String[] args){
        int i;
        int[] arr = new int[15];
        System.out.print("original: ");
        for(i=0;i<arr.length;i++){
            arr[i] = (int)(Math.random() * 1024);
            System.out.print(arr[i] + " ");
        }
        radixSort(arr);
        System.out.print("\nsorted: ");
        for(i=0;i<arr.length;i++)
            System.out.print(arr[i] + " ");
        System.out.println("\nDone ;-)");
    }
}
```

Yeah, not exactly the most friendliest code I've written. A few things to mention about the code. One: it's NOT fast (far from it!). Two: It only works with positive integers. Sample output:

```
original: 1023 1007 583 154 518 671 83 98 213 564 572 989 241 150 64
sorted: 64 83 98 150 154 213 241 518 564 572 583 671 989 1007 1023
Done ;-)
```

I don't like this sort much, so, I won't talk much about it. However, a few words before we go on. This sort can be rather efficient in conjunction with other sorts. For example, you can use this sort to pre-sort the most significant bits, and then use insertion sort for the rest. Another very crucial thing to do to speed it up is to make the node pool and queues statically allocated (don't allocate them every time you call the function). And if you're sorting small numbers (i.e.: `1-256`), you can make it `4` times as fast by simply removing the outer loop.

This sort is not very expandable. You'd have problems making it work with anything other than numbers. Even adding negative number support isn't very straight forward. Anyway, I'm going to go and think up more reasons not to use this sort... and use something like quick-sort instead.

Improving Radix Sort...

A day after I wrote the above section, I realized I made a blunder. Not only does that suck, but it's slow and has tons of useless code in it as well! For example, I realized that I didn't really need to care about the node pool that much. Even if I didn't setup the node pool, things would still work, since node pool gets re-initialized every time though the loop (look at the code in the above article). I later realized that I don't even need a pointer to the next free node on the node pool! Since we are allocating a node per number, and we are doing that in a loop, we could just use the loop counter to point to our free node!

After I got the code down to a mere 14 lines, I still wasn't satisfied with the inner loop that searched for the last node on the queue. So, created another array, just to hold the backs of queues. That eliminated lots of useless operations, and increased speed quite a bit, especially for large arrays.

After all that, I've moved the static arrays out of the function (just for the fun of it), since I didn't want to allocate exactly the same arrays every single time.

With each and every improvement, I was getting code which sucked less and less. Later, I just reset the 32 bit size to 16 bit integer size (to make it twice as fast, since the test program only throws stuff as large as 1024). I didn't go as far as unrolling the loops, but for a performance needy job, that could provide a few extra cycles.

Anyway, I won't bore you any longer, and just give you the new and improved code (which should have been the first one you've saw, but... I was lazy, and didn't really feel like putting in lots of effort into radix sort).

Yeah fast on large arrays otherwise too much overhead

```
import java.lang.*;
import java.io.*;
```

```
public class pRadixSort{

    private static int q[],ql[];
    static{
        q = new int[256];
        ql = new int[256];
        for(int i=0;i<q.length;q[i++] = -1);
    }

    public static void radixSort(int[] arr){
        int i,j,k,l,np[][] = new int[arr.length][2];
        for(k=0;k<2;k++){
            for(i=0;i<arr.length;np[i][0]=arr[i],np[i++][1]=-1)
                if(q[j=((255<<(k<<3))&arr[i])>>(k<<3)]==-1)
                    ql[j] = q[j] = i;
                else
                    ql[j] = np[ql[j]][1] = i;
            for(l=q[i=j=0];i<q.length;q[i++]=-1)
                for(l=q[i];l!=-1;l=np[l][1])
                    arr[j++] = np[l][0];
        }
    }

    public static void main(String[] args){
        int i;
        int[] arr = new int[15];
        System.out.print("original: ");
        for(i=0;i<arr.length;i++){
            arr[i] = (int)(Math.random() * 1024);
            System.out.print(arr[i] + " ");
        }
        radixSort(arr);
        System.out.print("\nsorted: ");
        for(i=0;i<arr.length;i++)
            System.out.print(arr[i] + " ");
        System.out.println("\nDone ;-)");
    }
}
```

This code is so much better than the previous one, that I'd consider it as fast as quicksort (in some cases). Counting the looping, we can get an approximate idea of how fast is it... First, we have a loop which repeats 2 times (for 16 bit numbers), inside of it, we have 2 loops, both of which repeat on the order of N. So, on average, I'd say this code should be about 2*2*N, which is 4N... (not bad... <if I did it correctly>), imagine, if you have 1000 elements in the array, the sort will only go through about 4000 loops (for bubble sort, it would have to loop for 1,000,000 times!). True, these loops are a bit complicated compared to the simple bubble sort loop, but the magnitude is still huge.

Anyway, I should really get some sleep right about now (didn't get much sleep in the past few days due to this little 'thought' which kept me awake). Never settle for code which you truly believe sucks (go do something about it!).

Reading and Writing Trees (Serialization)...

interesting he did not just edit document

Reading and writing binary trees to/from files is not as simple as reading and writing arrays. They're not linear, and require some algorithm to accomplish the task.

One simple approach would be to convert the tree to a node-pool one, and simply save the node-pool, and the first node (or something along those lines). That's the approach taken by lots of programs in saving trees. Actually, most programs implement the tree as a node-pool from the start, so, the whole thing is much simpler.

But what do you do when you're not using node-pools, but would still like to save the tree (and it's structure)? You can use a technique known as serialization.

There are many ways of serializing a binary tree, and incidentally, I've lost the *"formal algorithm,"* so, I'll re-derive it again, maybe in a little bit different form. The basic idea is to first write the size of data inside the node, and then the data itself. If there is no data you write zero, and return (without writing the data). If you've written any data, you then recursively write the left and right children of that node. There can be many variations of this; the most obvious is switching the order of left and right child.

The reading process has to know the way in which the tree was written. It has to first read the size of the data, if the size is zero, it sets the pointer to null, and returns. If the size is not zero, it allocates a new node in the tree, and reads the data into that node. Then, it recursively, reads the left and then the right children for that node.

This simple algorithm is very effective, and can be used to read and write data of different sizes (i.e.: character strings). The explanation of it may not be too clear at first; that's why I've written some source to clear up the confusion.

well give some sample output

The source that follows creates a binary tree holding strings, saves the tree to a file, and later reads that file.

```
import java.lang.*;
import java.io.*;

public class pBinTreeStringWR{
    public pBinTreeStringWR left,right;
    public String data;

    public static String[] strings = {"one","two",
        "three","four","five","six","seven","eight",
        "nine","ten","zero","computer","mouse","screen",
        "laptop","book","decimal","binary","quake"};

    public static pBinTreeStringWR tree_AddString(
        pBinTreeStringWR r,String s){
        if(r == null){
            r = new pBinTreeStringWR();
            r.left = r.right = null;
            r.data = s;
        }else if(r.data.compareTo(s) < 0)
            r.right = tree_AddString(r.right,s);
        else
            r.left = tree_AddString(r.left,s);
        return r;
    }

    public static void tree_InOrderPrint(
        pBinTreeStringWR r){
        if(r != null){
            tree_InOrderPrint(r.left);
            System.out.print(" "+r.data);
            tree_InOrderPrint(r.right);
        }
    }

    public static void tree_FileWrite(
        pBinTreeStringWR r,
        DataOutputStream output) throws IOException{
        if(r != null){
            byte[] tmp = r.data.getBytes();
            output.writeInt(tmp.length);
            output.write(tmp);
            tree_FileWrite(r.left,output);
            tree_FileWrite(r.right,output);
        }else
            output.writeInt(0);
    }

    public static pBinTreeStringWR tree_FileRead(
        pBinTreeStringWR r,
        DataInputStream input) throws IOException{
        int n = input.readInt();
        if(n != 0){
            byte[] tmp = new byte[n];
            input.read(tmp);
            r = new pBinTreeStringWR();
            r.data = new String(tmp);
            r.left = tree_FileRead(r.left,input);
            r.right = tree_FileRead(r.right,input);
        }else
            r = null;
        return r;
    }

    public static boolean tree_Compare(
        pBinTreeStringWR a,pBinTreeStringWR b){
        if(a != null && b != null){
            return a.data.compareTo(b.data) == 0 &&
                tree_Compare(a.left,b.left) &&
                tree_Compare(a.right,b.right);
        }else if(a == null && b == null)
            return true;
        else
            return false;
    }

    public static void main(String[] args){
        File file = new File("pBinTreeStringWR.dat");
```

```
        pBinTreeStringWR read_tree = null,tree = null;
        System.out.print("inserting: ");

        for(int i=0;i<strings.length;i++){
            String s = new String(strings[i]);
            System.out.print(" "+s);
            tree = tree_AddString(tree,s);
        }

        System.out.print("\ntree: ");
        tree_InOrderPrint(tree);

        System.out.println("\nwriting to "+file);
        try{
            tree_FileWrite(tree,
                new DataOutputStream(
                new FileOutputStream(file)));
        }catch(IOException e){
            System.out.println(e);
        }

        System.out.println("reading from "+file);
        try{
            read_tree = tree_FileRead(read_tree,
                new DataInputStream(
                new FileInputStream(file)));
        }catch(IOException e){
            System.out.println(e);
        }

        System.out.print("read tree: ");
        tree_InOrderPrint(read_tree);

        if(tree_Compare(tree,read_tree))
            System.out.println(
                "\nThe two trees are identical.");
        else
            System.out.println(
                "\nThe two trees are different.");
        System.out.println("done ;-)");
    }
}
```

The program both illustrates the algorithm and tests it. `pBinTreeStringWR` class is itself a node. These nodes are manipulated using `static` methods. I did it this way to reduce the number of needed classes (combining the program class with the node class).

In the program, we start out by creating a `File` object with the file we're about to write and read. We then declare two trees, one named `tree`, the other named `read_tree`. We then loop to add `static` strings to the `tree`. The add function adds them in a binary search tree fashion; it is using the `Comparable` interface to do the comparisons.

Then we open a file, create a `DataOutputStream` object to that file, and write the tree to that file using `tree_FileWrite()` function. This `tree_FileWrite()` closely matches the algorithm described earlier. If the node is not `null`, it writes the `length` of the `String`, and then the `String` itself (converted to `bytes`). It then recursively writes the `left` and `right` children of that node. If the node is initially `null`, the function simply writes a zero for the `length`, and returns.

The program continues by reading that file it has just written. It stores the read tree in `read_tree` variable. The writing process, `tree_FileRead()` also closely matches the algorithm described earlier. It is basically the opposite of writing algorithm. We first read the `length` of the `String`, if it's zero, we set the node to `null`, and continue on. If the `length` is not zero, we read that number of `bytes`, and store them in that node's `data String`. We then continue recursively by reading the `left` and `right` children of that `tree`.

The program then calls a compare function, which compares the trees (including the tree structure). Tests indicate that the program is working correctly, since the test function always returns that the tree written, and the tree read are identical. In the middle of all this, there is an in-order print function, but I don't think I need to describe it since I assume you know the basics of trees. (Basically, go over the source and you'll understand the whole thing.) The output from the program follows.

```
inserting:  one two three four five six seven eight
nine ten zero computer mouse screen laptop book decimal binary quake

tree:  binary book computer decimal eight five four laptop mouse
nine one quake screen seven six ten three two zero

writing to pBinTreeStringWR.dat
```

✓ here we go

so by algotm restores it

```
reading from pBinTreeStringWR.dat

read tree:  binary book computer decimal eight five four laptop
mouse nine one quake screen seven six ten three two zero

The two trees are identical.
done ;-)
```

← test

Another useful trick that you can use when you are not reading or writing variable length data like strings is not to write the `length`, but simply some marker. For example, lets say your tree holds integers, and you know all integers are 32 bits. Thus, your `length` parameter can simply be a `boolean` value of whether there exists a next node or not. The next example illustrates this by storing a `boolean` value instead of the `length` parameter.

```
import java.lang.*;
import java.io.*;

public class pBinTreeIntegerWR{
    public pBinTreeIntegerWR left,right;
    public Integer data;

    public static pBinTreeIntegerWR tree_AddNumber(
        pBinTreeIntegerWR r,Integer n){
        if(r == null){
            r = new pBinTreeIntegerWR();
            r.left = r.right = null;
            r.data = n;
        }else if(r.data.compareTo(n) < 0)
            r.right = tree_AddNumber(r.right,n);
        else
            r.left = tree_AddNumber(r.left,n);
        return r;
    }

    public static void tree_InOrderPrint(
        pBinTreeIntegerWR r){
        if(r != null){
            tree_InOrderPrint(r.left);
            System.out.print(" "+r.data);
            tree_InOrderPrint(r.right);
        }
    }

    public static void tree_FileWrite(
        pBinTreeIntegerWR r,
        DataOutputStream output) throws IOException{
        if(r != null){
            output.writeBoolean(true);
            output.writeInt(r.data.intValue());
            tree_FileWrite(r.left,output);
            tree_FileWrite(r.right,output);
        }else
            output.writeBoolean(false);
    }

    public static pBinTreeIntegerWR tree_FileRead(
        pBinTreeIntegerWR r,
        DataInputStream input) throws IOException{
        if(input.readBoolean()){
            r = new pBinTreeIntegerWR();
            r.data = new Integer(input.readInt());
            r.left = tree_FileRead(r.left,input);
            r.right = tree_FileRead(r.right,input);
        }else
            r = null;
        return r;
    }

    public static boolean tree_Compare(
        pBinTreeIntegerWR a,pBinTreeIntegerWR b){
        if(a != null && b != null){
            return a.data.compareTo(b.data) == 0 &&
                tree_Compare(a.left,b.left) &&
                tree_Compare(a.right,b.right);
        }else if(a == null && b == null)
            return true;
        else
            return false;
    }
```

```
    public static void main(String[] args){
        File file = new File("pBinTreeIntegerWR.dat");
        pBinTreeIntegerWR read_tree = null,tree = null;
        System.out.print("inserting: ");
        for(int i=0;i<10;i++){
            Integer n = new Integer((int)(Math.random()*100));
            System.out.print(" "+n);
            tree = tree_AddNumber(tree,n);
        }
        System.out.print("\ntree: ");
        tree_InOrderPrint(tree);
        System.out.println("\nwriting to "+file);
        try{
            tree_FileWrite(tree,
                new DataOutputStream(
                new FileOutputStream(file)));
        }catch(IOException e){
            System.out.println(e);
        }

        System.out.println("reading from "+file);
        try{
            read_tree = tree_FileRead(read_tree,
                new DataInputStream(
                new FileInputStream(file)));
        }catch(IOException e){
            System.out.println(e);
        }

        System.out.print("read tree: ");
        tree_InOrderPrint(read_tree);

        if(tree_Compare(tree,read_tree))
            System.out.println(
                "\nThe two trees are identical.");
        else
            System.out.println(
                "\nThe two trees are different.");
        System.out.println("done ;-)");
    }
}
```

The program above is almost identical to the one before, except it stores `java.lang.Integer` objects instead of `java.lang.String` objects. Since both of these implement `java.lang.Comparable interface`, the functions look pretty much the same. The only functions which look different are the file reading and writing. You should quickly notice that no `length` parameter is involved, but a simple `boolean` value telling us whether there exists a next node. Output from the program above follows:

```
inserting:  29 59 25 16 32 43 68 32 8 43
tree:  8 16 25 29 32 32 43 43 59 68
writing to pBinTreeIntegerWR.dat
reading from pBinTreeIntegerWR.dat
read tree:  8 16 25 29 32 32 43 43 59 68
The two trees are identical.
done ;-)
```

The funny thing that I always save till the end is that in practice, you'll probably never use any of these approaches yourself. Java provides a fairly useful `java.io.Serializable` interface, which given a tree, will nicely store it into a file. Anyway, it does pay to know how these kinds of things are done...

Deleting items from a Binary Search Tree...

Every database system must provide for deletion of items. A Binary Search Tree is a very good option for implementing a database (fast searches, sorts, inserts, etc.). One problem that stands in the way though is that it's not very straight forward to delete an item from a database represented by a binary tree. More precisely, the problem is in maintaining the tree structure while the deletion process.

Deleting items from a Binary Search Tree can be rather tricky. On one hand, you'd like to remove the object from the tree, on the other, you don't want the process to destroy the tree structure. There are many algorithms for this, and they all vary in degree of simplicity and efficiency.

The simplest one (which we will not cover here), is to simply have a `boolean` variable inside a node. That `boolean` variable tells us if that node is valid or not. Whenever we want to delete that node, we simply set that valid variable to

`false`; making all the traversal functions simply skip that node. The actual removal can take place when the tree is rebuilt. This may seem like a waste of space, but most of the time, it's not (in today's world, several extra bytes inside the tree structure don't make much of a difference).

The above can actually be used in conjunction with a more complicated approach. For example, lets take a regular company database. Throughout the day, there are thousands of transactions, deletions, insertions, etc., all this is handled by the algorithm described above (a `boolean` valid variable). At the end of the day (night), when the system becomes free, the program does a routine clean-up of the tree from these *"deleted"* items. Since setting or unsetting a `boolean` variable can be fast, the database is really fast during the day, i.e.: when it matters.

Cleaning up a tree from these types of *"deleted"* items can be accomplished in many different ways. If there are too many of them, then it might be worthwhile to simply rebuild the tree from scratch (making sure it doesn't loose it's advantageous structure, i.e.: doesn't become a linked list). The program could also fire up a *more complicated* approach to delete each node individually. (While at it, the system could also be optimizing the tree structure ;-)

Now, what is that *"more complicated"* approach that I keep talking of? It is the approach of switching the node being deleted with the one currently in the tree, only at lower level. Deleting a node that has no children is pretty simple, we just remove that node (and set the parent's pointer to it to `null`). Deleting a node that has one child is also easy. We delete the node, and make it's parent point to that only child of the deleted node.

The problem comes when you try to delete a node which has two valid children. Which one do you pick to be it's successor (take it's parent's place)? Actually, in most cases, neither!

You have to realize that we're dealing with a binary search tree. Search trees have very specific properties. For example, if we need to remove a node, we look for nearest right child that doesn't have a left son (or nearest left child, that doesn't have a right son). We then replace that newly found node with the one we are trying to delete (and making sure that all the links go where they should). The process of deleting that right child with no left son actually involves a simple removal described above: i.e.: the node is simply replaced by it's only child (in this case the right child; since there is no left). Confusing? Lets jump into the code to clear it up...

```
import java.lang.*;
import java.io.*;

public class pBSTRemoveNode{

    public pBSTRemoveNode left,right;
    public Comparable data;

    public static pBSTRemoveNode tree_AddNumber(
        pBSTRemoveNode r,Comparable n){
        if(r == null){
            r = new pBSTRemoveNode();
            r.left = r.right = null;
            r.data = n;
        }else if(r.data.compareTo(n) < 0)
            r.right = tree_AddNumber(r.right,n);
        else if(r.data.compareTo(n) > 0)
            r.left = tree_AddNumber(r.left,n);
        return r;
    }

    public static pBSTRemoveNode tree_removeNumber(
        pBSTRemoveNode r,Comparable n){
        if(r != null){
            if(r.data.compareTo(n) < 0){
                r.right = tree_removeNumber(r.right,n);
            }else if(r.data.compareTo(n) > 0){
                r.left = tree_removeNumber(r.left,n);
            }else{
                if(r.left == null && r.right == null){
                    r = null;
                }else if(r.left != null && r.right == null){
                    r = r.left;
                }else if(r.right != null && r.left == null){
                    r = r.right;
                }else{
                    if(r.right.left == null){
                        r.right.left = r.left;
                        r = r.right;
                    }else{
                        pBSTRemoveNode q,p = r.right;
                        while(p.left.left != null)
                            p = p.left;
                        q = p.left;
                        p.left = q.right;
                        q.left = r.left;
```

```
                        q.right = r.right;
                        r = q;
                    }
                }
            }
        }
        return r;
    }

    public static void tree_InOrderPrint(
        pBSTRemoveNode r){
        if(r != null){
            tree_InOrderPrint(r.left);
            System.out.print(" "+r.data);
            tree_InOrderPrint(r.right);
        }
    }

    public static void main(String[] args){
        pBSTRemoveNode tree = null;
        int[] numbers = {56,86,71,97,82,99,65,36,16,10,28,52,46};
        System.out.print("inserting: ");
        for(int i = 0;i<numbers.length;i++){
            Integer n = new Integer(numbers[i]);
            System.out.print(" "+n);
            tree = tree_AddNumber(tree,n);
        }
        System.out.print("\ntree: ");
        tree_InOrderPrint(tree);
        for(int j = 0;j < numbers.length;j++){
            Integer n = new Integer(numbers[j]);
            System.out.print("\nremove: "+n+" tree: ");
            tree = tree_removeNumber(tree,n);
            tree_InOrderPrint(tree);
        }
        System.out.println("\ndone ;-)");
    }
}
```

If you look through the above code, you'll quickly realize that the most relevant function (to this section), is the `tree_removeNumber()`. It accepts a reference to the `root` of the tree, and a number to remove. Actually, the way it's implemented, it doesn't necessarily has to be a number. It could just as well be a `java.lang.String` object; anything that's implementing a `Comparable interface` will work. The title of the function is a bit misleading; telling you that you can only work with numbers.

The `main()` is rather straight forward; it first adds numbers to the tree, and then removes them, displaying what it does at each step. The `tree_AddNumber()` function will not be explained here (since it's already been explained somewhere within this document).

The `tree_removeNumber()` method is where most of the interesting action takes place. We first check to see if the root of the tree is not `null`, since if it is, we have nothing to remove, and we simply `return`. The next two `if()` and `else if()` statements do what is know as binary search. If the item that we're looking for is greater than the value of the current node, we recursively search the right child of the current node. If it's less than the value of the current node, then we recursively search the left child of the current node.

The remainder of the function is kind of a big `else` statement. Once we're there, it means we have found the item we were looking for. We start by first checking for simple cases, where the node we're removing has no children, or has only one child. If it has two valid children, we fall into another `else` statement.

Once we know it is not one of the simple cases, we have to do a bit of thinking. First, we check a bit easier case, where the right child of the node doesn't have a left child. If that's the case, we need go no further, we simply replace the removing node with it's right child, and make sure the links are not lost. If the right child has a left child, we have to loop to find the closest right child with no left child, and that's what that `while()` loop is doing. The moment we find that node we would like to put in place of the one being deleted, we remove the found node. This removal is rather simple, since the node only has a right child. We then simply replace the removed node with the new one, making sure we don't loose any links, and we are done.

Inside `main()`, I have picked numbers to work with, and to construct the tree (sample data). The numbers generate a pretty good binary tree. The removal starts with the root node, so, the example does quite extensive testing in all the cases described above. The output from the above program follows:

```
inserting:  56 86 71 97 82 99 65 36 16 10 28 52 46
tree:  10 16 28 36 46 52 56 65 71 82 86 97 99
remove: 56 tree:  10 16 28 36 46 52 65 71 82 86 97 99
remove: 86 tree:  10 16 28 36 46 52 65 71 82 97 99
```

```
remove: 71 tree:  10 16 28 36 46 52 65 82 97 99
remove: 97 tree:  10 16 28 36 46 52 65 82 99
remove: 82 tree:  10 16 28 36 46 52 65 99
remove: 99 tree:  10 16 28 36 46 52 65
remove: 65 tree:  10 16 28 36 46 52
remove: 36 tree:  10 16 28 46 52
remove: 16 tree:  10 28 46 52
remove: 10 tree:  28 46 52
remove: 28 tree:  46 52
remove: 52 tree:  46
remove: 46 tree:
done ;-)
```

Trace the output if you like, it's quite interesting. This type of removing actually improves the tree structure, it makes it shorter (decreases tree's depth), thus, making searches faster. One thing that I'd like to mention before we go on, is that this example is for JDK 1.2 or above. It will not compile (nor run) on anything less due to the fact that it uses the `java.util.Comparable interface`, which is not supported in earlier versions of the JDK. Anyway, I guess that's it for this topic.

Determining Tree Depth...

Tree related algorithms depend on tree depth being small (otherwise, they become linked list algorithms ;-). Determining what is the depth of a tree can sometimes lead to optimizations, and other interesting things like that. How then, do you determine tree depth?

The task seems rather simple, however, there are a few tricks involved. Since most trees are defined recursively, our algorithm will also be recursive. We will need a wrapper method to initialize the maximum depth to zero, and then, recursively go through the tree determining the depth at each node. We will use a counter to keep the count of the current depth. Whenever we enter a recursive method, we will increment the counter, whenever we leave, we will decrement it. If that counter is greater than the maximum tree depth encountered so far, we will set the maximum tree depth to the value of that counter.

did for 6.034

Once the algorithm completes, the variable holding the maximum tree depth will hold the max tree depth (sounds logical doesn't it?) Anyway, talk is cheap, lets go write it!

```
import java.lang.*;
import java.io.*;

public class pBinTreeDepth{
    public pBinTreeDepth left,right;
    public Integer data;
    private static int tree_depth,curr_depth = 0;
    public static int[] numbers = {7,3,11,2,5,9,12,4,6,8,10};

    public static pBinTreeDepth add(pBinTreeDepth r,Integer n){
        if(r == null){
            r = new pBinTreeDepth();
            r.left = r.right = null;
            r.data = n;
        }else if(r.data.compareTo(n) < 0)
            r.right = add(r.right,n);
        else
            r.left = add(r.left,n);
        return r;
    }

    public static void print(pBinTreeDepth r){
        if(r != null){
            print(r.left);
            System.out.print(" "+r.data);
            print(r.right);
        }
    }

    public static void _getdepth(pBinTreeDepth r){
        if(r != null){
            curr_depth++;
            if(curr_depth > tree_depth)
                tree_depth = curr_depth;
            _getdepth(r.left);
            _getdepth(r.right);
            curr_depth--;
        }
    }
```

```
    public static int getdepth(pBinTreeDepth r){
        tree_depth = 0;
        _getdepth(r);
        return tree_depth;
    }

    public static void main(String[] args){
        pBinTreeDepth tree = null;
        System.out.print("inserting: ");
        for(int i=0;i<numbers.length;i++){
            Integer n = new Integer(numbers[i]);
            System.out.print(" "+n);
            tree = add(tree,n);
        }
        System.out.print("\ntree: ");
        print(tree);
        System.out.println("\ndepth: "+getdepth(tree));
        System.out.println("done ;-)");
    }
}
```

The code above creates a binary search tree, and then calls the `getdepth(pBinTreeDepth)` method to get the tree's depth. This method in turn calls `_getdepth(pBinTreeDepth)`, which is the actual recursive procedure. The whole thing is implemented as `static` methods; this is just to make quick & simple implementation easier.

Advanced Linked Lists...

We have already talked about linked lists previously in this document, and the assumption was that you'll learn how to use them. We didn't however explain a lot of the more used types of lists (not many people use a plain old linked list). As it stands, the linked list explained earlier is pretty bad and inefficient.

The most critical problem is that insertions and deletions from the tail of the list are slow. To delete something from the end, we would have to loop until we hit the end, and only then remove the item. This limitation is fairly obvious, since it would be extremely inefficient to use that kind of a list (single linked) to implement a queue.

The solution is to use a doubly linked list; where each node has two pointers, one to it's left neighbor, and one to it's right, and to have a `head` and a `tail` pointer. We will later implement this type of a list.

Another critical problem with the previous lists is that they have special cases in insert and remove methods. Ideally, we would just like to insert and/or delete, without any special cases (like `null` head pointer). How can we speed-up, and simplify the insertion and deletion operations? Easy! We simply have a dummy `head` pointer which is always there (but is not storing any data). Since we're interested in doubly linked lists, we would have two dummy pointers; the dummy `head`, and dummy `tail`.

Some may argue that having nodes that store no data is useless, and wastes memory. That may be true for some cases, where memory is critical, but for most purposes, the speed and simplicity gained greatly outweigh the wasted memory disadvantage.

You should still keep in mind the above. Sometimes, you end up with arrays of arrays of arrays of linked lists, and in those cases, those few wasted bytes, can add up to hundreds of megabytes. For example, it's pointless to have dummy pointers if the lists never get beyond two or three elements. Before implementing *anything*, think about the approach you're using.

Another strange thing our previous lists had was a `peek(int)` method. We used it to go through the list, and view it's contents. This may have worked quite well when the list was implemented using an array (where we directly jump to that location), but when we were using linked lists, this `peek(int)` procedure got quite slow. Given that we're looping through every element, and every time, we have to loop until we hit that number inside the list, it starts to become obvious that it's a waste of time.

What can we use to go through the items in the list, do it safely, and more efficiently? In C++ world, programmers are quite familiar in writing iterators. Iterators are used to iterate through objects contained in some data structure (usually some data container class). In Java, we have something similar available to us. It is called the `Enumeration`. Java provides the standard `java.util.Enumeration` object for us to use to go through the items in the list. In fact, because the `java.util.Enumeration` is standard, even `java.util.Vector` class uses it!

So, whenever we need to iterate through every element in the list, we simply get the `Enumeration` for the class, and use that to go through the elements. Details of the implementation are described later.

For now, we have improved our view of the list quite a bit. You should still never forget to be inventive. There are other ways to represent a linked list (for example, make it circular, with `head` and `tail` being the same node). Hopefully,

we'll later examine tree representation of a linked list. A tree representation gives you the best of both worlds, linked structure, and fast insertions and deletions (more on that later, hopefully).

Doubly Linked Lists (with Enumeration)...

Doubly linked lists are not much different from singly linked lists; we just have an extra pointer to worry about. As usual, if you don't understand something, it help to draw it out on paper. Yeah, go ahead and draw a linked list, then go through the operations by drawing or erasing links.

Anyway, lets get right to the point, and write it.

```
import java.lang.String;
import java.io.*;
import java.util.*;
import pTwoChildNode;

public class pDoublyLinkedList{

    private pTwoChildNode head,tail;
    protected long num;

    protected pTwoChildNode getHead(){
        return head;
    }

    protected pTwoChildNode getTail(){
        return tail;
    }

    protected void setHead(pTwoChildNode p){
        head = p;
    }

    protected void setTail(pTwoChildNode p){
        tail = p;
    }

    public pDoublyLinkedList(){
        setHead(new pTwoChildNode());
        setTail(new pTwoChildNode());
        getTail().setLeft(head);
        getHead().setRight(tail);
        num = 0;
    }

    public long size(){
        return num;
    }

    public boolean isEmpty(){
        return num == 0;
    }

    public void addHead(Object o){
        pTwoChildNode p = new pTwoChildNode(o);
        p.setLeft(getHead());
        p.setRight(getHead().getRight());
        getHead().setRight(p);
        p.getRight().setLeft(p);
        num++;
    }

    public Object removeHead(){
        Object o = null;
        if(!isEmpty()){
            pTwoChildNode p = getHead().getRight();
            getHead().setRight(p.getRight());
            p.getRight().setLeft(getHead());
            o = p.getData();
            num--;
        }
        return o;
    }

    public void addTail(Object o){
        pTwoChildNode p = new pTwoChildNode(o);
```

```
        p.setRight(getTail());
        p.setLeft(getTail().getLeft());
        getTail().setLeft(p);
        p.getLeft().setRight(p);
        num++;
    }

    public Object removeTail(){
        Object o = null;
        if(!isEmpty()){
            pTwoChildNode p = getTail().getLeft();
            getTail().setLeft(p.getLeft());
            p.getLeft().setRight(getTail());
            o = p.getData();
            num--;
        }
        return o;
    }

    public void add(Object o){
        addHead(o);
    }

    public Object remove(){
        return removeHead();
    }

    public Enumeration elementsHeadToTail(){
        return new Enumeration(){

            pTwoChildNode p = getHead();

            public boolean hasMoreElements(){
                return p.getRight() != getTail();
            }

            public Object nextElement(){
                synchronized(pDoublyLinkedList.this){
                    if(hasMoreElements()){
                        p = p.getRight();
                        return p.getData();
                    }
                }
                throw new NoSuchElementException(
                    "pDoublyLinkedList Enumeration");
            }
        };
    }

    public Enumeration elementsTailToHead(){
        return new Enumeration(){

            pTwoChildNode p = getTail();

            public boolean hasMoreElements(){
                return p.getLeft() != getHead();
            }

            public Object nextElement(){
                synchronized(pDoublyLinkedList.this){
                    if(hasMoreElements()){
                        p = p.getLeft();
                        return p.getData();
                    }
                }
                throw new NoSuchElementException(
                    "pDoublyLinkedList Enumeration");
            }
        };
    }

    public static void main(String[] args){
        pDoublyLinkedList list = new pDoublyLinkedList();
        int i;
        System.out.println("inserting head:");
        for(i=0;i<5;i++){
            Integer n = new Integer((int)(Math.random()*99));
            list.addHead(n);
            System.out.print(n+" ");
```

```
        }
        System.out.println("\ninserting tail:");
        for(i=0;i<5;i++){
            Integer n = new Integer((int)(Math.random()*99));
            list.addTail(n);
            System.out.print(n+" ");
        }
        System.out.println("\nhead to tail print...");
        Enumeration enum = list.elementsHeadToTail();
        while(enum.hasMoreElements())
            System.out.print(((Integer)enum.nextElement())+" ");
        System.out.println("\ntail to head print...");
        enum = list.elementsTailToHead();
        while(enum.hasMoreElements())
            System.out.print(((Integer)enum.nextElement())+" ");
        System.out.println("\nremoving head:");
        for(i=0;i<5;i++){
            Integer n = (Integer)list.removeHead();
            System.out.print(n+" ");
        }
        System.out.println("\nremoving tail:");
        while(!list.isEmpty()){
            Integer n = (Integer)list.removeTail();
            System.out.print(n+" ");
        }
        System.out.println("\ndone ;-)");
    }
}
```

The above code both implements the doubly linked list, and tests it. The code also uses `pTwoChildNode` object we've developed earlier. (It is simply a node with two children ;-) Output from the above program follows:

```
inserting head:
0 39 33 14 51
inserting tail:
42 25 76 43 56
head to tail print...
51 14 33 39 0 42 25 76 43 56
tail to head print...
56 43 76 25 42 0 39 33 14 51
removing head:
51 14 33 39 0
removing tail:
56 43 76 25 42
done ;-)
```

You can try tracing the output (it helps sometimes), or you can just look at the source and see what's happening. The testing procedure should seem like second nature by this time...

The list is built around two dummy nodes, the `head` and `tail`. These are created at the time of the constructor call, and remain valid until the class gets swept away by garbage collection (when it goes out of scope). The `addHead()` method simply inserts the new node right after the `head` dummy node, and `addTail()` right before the `tail` node. The remove functions do their appropriate functions.

There really isn't much to explain; just look at the source, and you'll figure it out (there is nothing here more complex than what you've already seen). What you should be curious about is the `elementsHeadToTail()` and `elementsTailToHead()` methods. These methods return an `Enumeration` object of the `java.util` package.

The `Enumeration` object is created (and declared), inside the functions! (don't you just love Java?) All this object contains is a pointer to a node inside the list. The `java.util.Enumeration` interface has two functions, and we simply use these two functions to make it possible to step through the elements of the list. If you've ever used iterators in C++, or used `java.util.Enumeration` object with `java.util.Vector`, then this should be pretty easy to comprehend.

This step-through-the-list method is fairly safe, since we're not giving away the safety of our list structure, and we're controlling everything (the user can't just access protected members of the list class, yet the user gets a fairly fast and efficient way to loop through every element of the list as if they were able to access the inside elements of the list.) This code is much more superior to the `Object peek(int)` method we used in previous lists.

The main point of this section was not to show you one particular implementation, but to help you realize that linked lists are much more flexible than it is evident from their first appearance.

Binary Space Partition (BSP) Trees...

- Dog3D DEMO - (Click here to see the demo for this section)

As mentioned previously, tree data structures are wonderful in some cases for certain purposes. One of these purposes is Hidden Surface Removal (HSR). HSR in graphics turns out to be quite a complicated problem. To paraphrase one book, "HSR is a thorn in a graphics programmer's back".

Most graphics programmers today are more concerned with an efficient way of eliminating hidden (or unseen) surfaces, than with the inner workings of pixel plotting and/or texture mapping. Games like *Wolfenstain 3D*, *DOOM*, and *Quake* by *id Software*, are mostly reflections of innovations in the field of HSR. (and a bit of added processing power ;-)

First, lets define HSR in a more understandable manner. Imagine you're standing in a room, you can only see the walls of the room you're in, and not the walls of other rooms (which are beside your room). The walls of your room cover up your view, so, you can't see anything else other than the walls of your room. This relatively simple concept turns out to be quite a bundle when it comes to computers. There are literally hundreds of different approaches to this, and they all have their advantages and disadvantages.

One solution used in *Wolfenstain 3D* is *ray casting*. Ray casting simply passes a horizontal *"ray"* (or two rays) across a map (a map in such a case is simply a 2D array of values representing blocks); if a ray hits a *"filled"* block on the map, then a vertical scaled texture is drawn, if not, the ray continues on to the next block. This produces a pretty blocky world, evidenced by *Wolfenstain 3D*.

Another solution is *ray tracing*, it's a bit more involved than *ray casting* (actually, *ray casting* came from simplifying *ray tracing*). In this, a ray is passed over all pixels on the screen, and when a ray hits an object in 3D space, that pixel is drawn having a texture color of that object. This sounds like a lot of work, and it is. Ray tracing, currently, is only good for high quality images, and not for real time games where images are generated on the fly. (it can easily take minutes or even hours to ray trace a scene)

There are many others, like Z-Buffering, Painter's Algorithm, Portals, etc., and the one we're here to talk about is Binary Space Partition (BSP). BSP was used successfully in games like DOOM and Quake, and has the potential for a lot more. It is a process of recursively subdividing space, building a binary tree, and later traversing the tree, knowing what to draw and when.

Imagine for example that you had to draw two rooms, one beside the other, with a small door in between. What you can do is draw the walls of the farther room, then draw the door, and draw the walls of the room you're in, overwriting parts of already drawn walls of the farther room. Now, how can you figure out where you're located (in which room), so that you can first draw the farther room, and then draw the room you're in? Easy, you create a binary tree of the world; then, given your point in space, you can easily determine your location relative to the world. Thus, you can determine what to draw first, and what to draw later. (to produce a nice, real looking 3D world)

To understand this concept you need to be able to visualize the tree, and how you traverse it (have a solid understanding of the tree structure). First, you create a tree of the world, by selecting a line (or plane in 3D), adding that line (or plane) to this node; you later use that line (or plane), to sort the whole world into two. One side with lines (or planes) that are in "front" of that selected line (or plane), and the rest which are in the back. The front and back are determined from the line (or plane) equation, and the x, y, z parameters from the lines (or planes) being checked. If there comes a time where some line (or plane) is neither in front or in back (has points on both sides), it is split (partitioned) into two, one side goes in front, and the other side in back. The process recursively continues with each of these new subsets until there are no more lines to select.

The result is a tree representing the world. To traverse it, you evaluate the player's location in relation to the root node, if it is in front, you recursively draw the back, and then the front, if it is in back, you recursively draw the front, and then the back. This simple procedure produces a nicely sorted list of *"walls"* to draw. The above describes a back to front traversal, you do totally the opposite for a front to back traversal, which seems to be getting more popular now a days.

In a back to front traversal, you don't have to worry about clipping; everything looks perfect after drawing, since everything unwanted is overwritten (i.e.: Painter's Algorithm). In a front to back traversal, you've got quite a lot to worry about because of clipping. You need an efficient way of remembering which pixels have been drawn, etc. For now, we'll be mostly concerned with back to front traversal because of it's simple nature.

In this type of a discussion, it helps to keep things simple; I will only describe how to do this type of a thing for a very primitive 2D case. We will take a bunch of line coordinates, create a binary space partition tree, and later traverse that tree, displaying a 3D looking world (which is actually 2D). Don't feel too bad. 2D is simple and lets you understand the structure. DOOM is totally 2D, and still looks really cool. 3D is full of math problems which will only complicate the matter at this point. Besides, once you understand the structure, writing your own 3D implementation shouldn't be a problem (if you can get through the 3D math ;-)

Well, lets get to it. The first thing that we need for any kind of tree structure is a node. In our case, the node should contain the partition plane (in our case the line equation of a line dividing this node), and a list of lines currently spanning this node. Of course, it wouldn't be a tree node without at least two pointers to it's children; so, we'll include those too! Follows the source for this simple, yet useful node.

```
class javadata_dog3dBSPNode{
```

```
    public float[] partition = null;
    public Object[] lines = null;
    public javadata_dog3dBSPNode front = null;
    public javadata_dog3dBSPNode back = null;
}
```

As you can see, there is nothing tricky or hard to the piece of code above. Right now is a good time to figure out the conventions used in this program. A point is represented by a two element integer array. A line is represented by a five element integer array; the first four are for the starting point and ending point respectively, and the last is for the color. A partition plane (line equation) is represented by a three element floating point array. Because our partition planes are not normalized, we could as well used an integer array, but I doubt the several floating point calculations would have made much difference (even for a below-Pentium system!).

You'll also notice that the class above is not public, that's because all the Dog 3D classes are contained within one file (*javadata_dog3d.java* in case you're interested). (this program is not very modular, and separate parts make little sense outside of the program)

What we need next is our Binary Space Partition Tree to manipulate the nodes we've just created. The tree should be able to accept a list of lines, and build itself. It should also be able to traverse itself (in our case render itself). And lastly, it should contain (or have access to) all the necessary methods for working with points and lines (i.e.: comparison functions, splitting functions, etc.). The source for the Binary Space Partition Tree follows:

```
class javadata_dog3dBSPTree{
    private javadata_dog3dBSPNode root;
    public int eye_x,eye_y;
    public double eye_angle;
    private javadata_dog3d theParent = null;

    private final static int SPANNING = 0;
    private final static int IN_FRONT = 1;
    private final static int IN_BACK = 2;
    private final static int COINCIDENT = 3;

    public javadata_dog3dBSPTree(javadata_dog3d p){
        root = null;
        eye_x = eye_y = 0;
        eye_angle = 0.0;
        theParent = p;
    }

    private float[] getLineEquation(int[] line){
        float[] equation = new float[3];
        int dx = line[2] - line[0];
        int dy = line[3] - line[1];
        equation[0] = -dy;
        equation[1] = dx;
        equation[2] = dy*line[0] - dx*line[1];
        return equation;
    }

    private int evalPoint(int x,int y,float[] p){
        double c = p[0]*x + p[1]*y + p[2];
        if(c > 0)
            return IN_FRONT;
        else if(c < 0)
            return IN_BACK;
        else return SPANNING;
    }

    private int evalLine(int[] line,float[] partition){
        int a = evalPoint(line[0],line[1],partition);
        int b = evalPoint(line[2],line[3],partition);
        if(a == SPANNING){
            if(b == SPANNING)
                return COINCIDENT;
            else return b;
        }if(b == SPANNING){
            if(a == SPANNING)
                return COINCIDENT;
            else return a;
        }if((a == IN_FRONT) && (b == IN_BACK))
            return SPANNING;
        if((a == IN_BACK) && (b == IN_FRONT))
            return SPANNING;
        return a;
    }
```

```
    public int[][] splitLine(int[] l,float[] p){
        int[][] q = new int[2][5];
        q[0][4] = q[1][4] = l[4];
        int cross_x = 0,cross_y = 0;
        float[] lEq = getLineEquation(l);
        double divider = p[0] * lEq[1] - p[1] * lEq[0];
        if(divider == 0){
            if(lEq[0] == 0)
                cross_x = l[0];
            if(lEq[1] == 0)
                cross_y = l[1];
            if(p[0] == 0)
                cross_y = (int)-p[1];
            if(p[1] == 0)
                cross_x = (int)p[2];
        }else{
            cross_x = (int)((-p[2]*lEq[1] + p[1]*lEq[2])/divider);
            cross_y = (int)((-p[0]*lEq[2] + p[2]*lEq[0])/divider);
        }
        int p1 = evalPoint(l[0],l[1],p);
        int p2 = evalPoint(l[2],l[3],p);
        if((p1 == IN_BACK) && (p2 == IN_FRONT)){
            q[0][0] = cross_x;  q[0][1] = cross_y;
            q[0][2] = l[2];     q[0][3] = l[3];
            q[1][0] = l[0];     q[1][1] = l[1];
            q[1][2] = cross_x;  q[1][3] = cross_y;
        }else if((p1 == IN_FRONT) && (p2 == IN_BACK)){
            q[0][0] = l[0];     q[0][1] = l[1];
            q[0][2] = cross_x;  q[0][3] = cross_y;
            q[1][0] = cross_x;  q[1][1] = cross_y;
            q[1][2] = l[2];     q[1][3] = l[3];
        }else
            return null;
        return q;
    }

    private void build(javadata_dog3dBSPNode tree,Vector lines){
        int[] current_line = null;
        Enumeration enum = lines.elements();
        if(enum.hasMoreElements())
            current_line = (int[])enum.nextElement();
        tree.partition = getLineEquation(current_line);
        Vector _lines = new Vector();

        _lines.addElement(current_line);
        Vector front_list = new Vector();
        Vector back_list = new Vector();
        int[] line = null;
        while(enum.hasMoreElements()){
            line = (int[])enum.nextElement();
            int result = evalLine(line,tree.partition);
            if(result == IN_FRONT)            /* in front */
                front_list.addElement(line);
            else if(result == IN_BACK)        /* in back */
                back_list.addElement(line);
            else if(result == SPANNING){      /* spanning */
                int[][] split_line = splitLine(line,tree.partition);
                if(split_line != null){
                    front_list.addElement(split_line[0]);
                    back_list.addElement(split_line[1]);
                }else{
                    /* error here! */
                }
            }else if(result == COINCIDENT)
                _lines.addElement(line);
        }
        if(!front_list.isEmpty()){
            tree.front = new javadata_dog3dBSPNode();
            build(tree.front,front_list);
        }if(!back_list.isEmpty()){
            tree.back = new javadata_dog3dBSPNode();
            build(tree.back,back_list);
        }
        tree.lines = new Object[_lines.size()];
        _lines.copyInto(tree.lines);
    }

    public void build(Vector lines){
        if(root == null)
```

```
            root = new javadata_dog3dBSPNode();
        build(root,lines);
    }

    private void renderTree(javadata_dog3dBSPNode tree){
        int[] tmp = null;
        if(tree == null)
            return; /* check for end */
        int i,j = tree.lines.length;
        int result = evalPoint(eye_x,eye_y,tree.partition);
        if(result == IN_FRONT){
            renderTree(tree.back);
            for(i=0;i<j;i++){
                tmp = (int[])tree.lines[i];
                if(evalPoint(eye_x,eye_y,getLineEquation(tmp)) == IN_FRONT)
                    theParent.renderLine(tmp);
            }
            renderTree(tree.front);
        }else if(result == IN_BACK){
            renderTree(tree.front);
            for(i=0;i<j;i++){
                tmp = (int[])tree.lines[i];
                if(evalPoint(eye_x,eye_y,getLineEquation(tmp)) == IN_FRONT)
                    theParent.renderLine(tmp);
            }
            renderTree(tree.back);
        }else{    /* the eye is on the partition plane */
            renderTree(tree.front);
            renderTree(tree.back);
        }
    }

    public void renderTree(){
        renderTree(root);
    }
}
```

The above might look intimidating, but it's actually really simple. The are a lot of data members; some familiar, some are not. The `root` data member is obvious, it's the `root` of the tree! The next several are the eye's current position. We need this since we don't want to pass them as a parameter every time we render. The next data member is `theParent`, which is a reference back to the original applet. This member is used during the rendering process (not very modular). The last data members are constants for the point and line comparison functions.

The constructor takes the parent applet as it's parameter, and initializes `theParent` and other data members. The actual insertion of data into the tree is accomplished with the `build(java.util.Vector)` method. This method calls a more complicated method of the same name, but with more parameters. The `build()` method goes through the given `java.util.Vector`. It first selects the first line in the list to be the splitting plane (for the current node). It then goes through the rest of the list, sorting the lines in relation to that splitting plane. If a line is in front (determined by the evaluation functions) then it's added to the front list. If a line is in back, it's added to the back list. If a line is spanning the splitting node (has end points on both sides of the splitting node), then that line is split by a `splitLine()` function; one part goes in front, and the other into the back. If some line is actually coincident with the splitting plane, then it's added to the list of the current node. After all that, we end up with two lists of lines; one list for the back, and one list for the front. We then recursively go through the two lists.

The process described above is fairly hard to describe (an oxymoron?). If you want a more formal (*maybe better?*) description, you can search the Internet for the *"BSPFAQ."* It is a document thoroughly describing BSP trees in a more formal way.

Once the tree is built, you are ready to traverse it! The traversing is accomplished by calling the `renderTree()` method. What it does is first evaluate the eye's position in relation to the current splitting plane of the node. If it's in `front`, we recursively render the `back` child, if it's in `back`, we recursively render the `front` child. The rendering itself is accomplished by looping through the lines of the current node and drawing them. The evaluation function call inside that loop is doing back-face-culling (back-face-removal). It is a process of making sure that we're not drawing lines (walls) which are not facing us. The drawing is done by calling `renderLine()` method of `theParent` (which is a reference back to the original applet).

If you've survived this far, you're in good shape. The remaining part of this applet is simply the initialization and rendering, which is not really related to data structures. Anyway, here we go again, diving into some source before explaining it...

```
public class javadata_dog3d extends Applet implements Runnable{
    private Thread m_dog3d = null;
    private String m_map = "javadata_dog3dmap.txt";
    private int m_width,m_height;
    private int m_mousex = 0,m_mousey = 0;
```

```
    private Vector initial_map = null;
    private javadata_dog3dBSPTree theTree = null;
    private Image double_image = null;
    private Graphics double_graphics = null;
    public int eye_x = 220,eye_y = 220;
    public double eye_angle = 0;
    private boolean KEYUP=false,KEYDOWN=false,
        KEYLEFT=false,KEYRIGHT=false;
    private boolean MOUSEUP=true;

    public void renderLine(int[] l){
        double x1=l[2];
        double y1=l[3];
        double x2=l[0];
        double y2=l[1];
        double pCos = Math.cos(eye_angle);
        double pSin = Math.sin(eye_angle);
        int[] x = new int[4];
        int[] y = new int[4];
        double pD=-pSin*eye_x+pCos*eye_y;
        double pDp=pCos*eye_x+pSin*eye_y;
        double rz1,rz2,rx1,rx2;
        int Screen_x1=0,Screen_x2=0;
        double Screen_y1,Screen_y2,Screen_y3,Screen_y4;
        rz1=pCos*x1+pSin*y1-pDp;      //perpendicular line to the players
        rz2=pCos*x2+pSin*y2-pDp;      //view point
        if((rz1<1) && (rz2<1))
            return;
        rx1=pCos*y1-pSin*x1-pD;
        rx2=pCos*y2-pSin*x2-pD;
        double pTan = 0;
        if((x2-x1) == 0)
            pTan = Double.MAX_VALUE;
        else
            pTan = (y2-y1)/(x2-x1);
        pTan = (pTan-Math.tan(eye_angle))/(1+
            (pTan*Math.tan(eye_angle)));
        if(rz1 < 1){
            rx1+=(1-rz1)*pTan;
            rz1=1;
        }if(rz2 < 1){
            rx2+=(1-rz2)*pTan;
            rz2=1;
        }
        double z1 = m_width/2/rz1;
        double z2 = m_width/2/rz2;
        Screen_x1=(int)(m_width/2-rx1*z1);
        Screen_x2=(int)(m_width/2-rx2*z2);
        if(Screen_x1 > m_width)
            return;
        if(Screen_x2<0)
            return;
        int wt=88;
        int wb=-40;
        Screen_y1=(double)m_height/2-(double)wt*z1;
        Screen_y4=(double)m_height/2-(double)wb*z1;
        Screen_y2=(double)m_height/2-(double)wt*z2;
        Screen_y3=(double)m_height/2-(double)wb*z2;
        if(Screen_x1 < 0){
            Screen_y1+=(0-Screen_x1)*(Screen_y2-Screen_y1)
                /(Screen_x2-Screen_x1);
            Screen_y4+=(0-Screen_x1)*(Screen_y3-Screen_y4)
                /(Screen_x2-Screen_x1);
            Screen_x1=0;
        }if(Screen_x2 > (m_width)){
            Screen_y2-=(Screen_x2-m_width)*(Screen_y2-Screen_y1)
                /(Screen_x2-Screen_x1);
            Screen_y3-=(Screen_x2-m_width)*(Screen_y3-Screen_y4)
                /(Screen_x2-Screen_x1);
            Screen_x2=m_width;
        }if((Screen_x2-Screen_x1) == 0)
            return;
        x[0] = (int)Screen_x1;
        y[0] = (int)(Screen_y1);
        x[1] = (int)Screen_x2;
        y[1] = (int)(Screen_y2);
        x[2] = (int)Screen_x2;
        y[2] = (int)(Screen_y3);
        x[3] = (int)Screen_x1;
```

```
        y[3] = (int)(Screen_y4);
        double_graphics.setColor(new Color(l[4]));
        double_graphics.fillPolygon(x,y,4);
    }

    private void loadInputMap(){
        initial_map = new Vector();
        int[] tmp = null;
        int current;
        StreamTokenizer st = null;
        try{
            st = new StreamTokenizer(
                (new URL(getDocumentBase(),m_map)).openStream());
        }catch(java.net.MalformedURLException e){
            System.out.println(e);
        }catch(IOException f){
            System.out.println(f);
        }
        st.eolIsSignificant(true);
        st.slashStarComments(true);
        st.ordinaryChar('\'');
        try{
            for(st.nextToken(),tmp = new int[5],current=1;
            st.ttype != StreamTokenizer.TT_EOF;
            st.nextToken(),current++){
                if(st.ttype == StreamTokenizer.TT_EOL){
                    if(tmp != null)
                        initial_map.addElement(tmp);
                    tmp = null; tmp = new int[5];
                    current=0;
                }else{
                    if(current == 1)
                        System.out.println("getting: "+st.nval);
                    else if(current == 2)
                        tmp[0] = (int)st.nval;
                    else if(current == 3)
                        tmp[1] = (int)st.nval;
                    else if(current == 4)
                        tmp[2] = (int)st.nval;
                    else if(current == 5)
                        tmp[3] = (int)st.nval;
                    else if(current == 6)
                        tmp[4] = (int)Integer.parseInt(st.sval,0x10);
                }
            }
        }catch(IOException e){
            System.out.println(e);
        }
    }

    public void init(){
        String param;
        param = getParameter("map");
        if (param != null)
            m_map = param;
        m_width = size().width;
        m_height = size().height;

        loadInputMap();

        double_image = createImage(m_width,m_height);
        double_graphics = double_image.getGraphics();

        theTree = new javadata_dog3dBSPTree(this);
        theTree.build(initial_map);

        theTree.eye_x = eye_x;
        theTree.eye_y = eye_y;
        theTree.eye_angle = eye_angle;

        repaint();
    }

    public void paint(Graphics g){
        g.drawImage(double_image,0,0,null);
    }

    public void update(Graphics g){
        double_graphics.setColor(Color.black);
```

```
        double_graphics.fillRect(0,0,m_width,m_height);
        theTree.renderTree();
        paint(g);
    }

    public void start(){
        if(m_dog3d == null){
            m_dog3d = new Thread(this);
            m_dog3d.start();
        }
    }

    public void run(){
        boolean call_update;
        while(true){
            call_update = false;
            if(MOUSEUP){
                if(KEYUP){
                    eye_x += (int)(Math.cos(eye_angle)*10);
                    eye_y += (int)(Math.sin(eye_angle)*10);
                    call_update = true;
                }if(KEYDOWN){
                    eye_x -= (int)(Math.cos(eye_angle)*10);
                    eye_y -= (int)(Math.sin(eye_angle)*10);
                    call_update = true;
                }if(KEYLEFT){
                    eye_angle += Math.PI/45;
                    call_update = true;
                }if(KEYRIGHT){
                    eye_angle -= Math.PI/45;
                    call_update = true;
                }if(call_update){
                    theTree.eye_x = eye_x;
                    theTree.eye_y = eye_y;
                    theTree.eye_angle = eye_angle;
                    repaint();
                }
            }
            try{
                Thread.sleep(5);
            }catch(java.lang.InterruptedException e){
                System.out.println(e);
            }
        }
    }

    public boolean keyUp(Event evt,int key){
        if(key == Event.UP){
            KEYUP = false;
        }else if(key == Event.DOWN){
            KEYDOWN = false;
        }else if(key == Event.LEFT){
            KEYLEFT = false;
        }else if(key == Event.RIGHT){
            KEYRIGHT = false;
        }
        return true;
    }

    public boolean keyDown(Event evt,int key){
        if(key == Event.UP){
            KEYUP = true;
        }else if(key == Event.DOWN){
            KEYDOWN = true;
        }else if(key == Event.LEFT){
            KEYLEFT = true;
        }else if(key == Event.RIGHT){
            KEYRIGHT = true;
        }
        return true;
    }

    public boolean mouseDown(Event evt, int x, int y){
        MOUSEUP = false;
        m_mousex = x;
        m_mousey = y;
        return true;
    }
```

```
        public boolean mouseUp(Event evt, int x, int y){
            MOUSEUP = true;
            m_mousex = x;
            m_mousey = y;
            return true;
        }

        public boolean mouseDrag(Event evt, int x, int y){
            if(m_mousey > y){
                eye_x += (int)(Math.cos(eye_angle)*(7));
                eye_y += (int)(Math.sin(eye_angle)*(7));
            }
            if(m_mousey < y){
                eye_x -= (int)(Math.cos(eye_angle)*(7));
                eye_y -= (int)(Math.sin(eye_angle)*(7));
            }
            if(m_mousex > x){
                eye_angle += Math.PI/32;
            }
            if(m_mousex < x){
                eye_angle -= Math.PI/32;
            }
            theTree.eye_x = eye_x;
            theTree.eye_y = eye_y;
            theTree.eye_angle = eye_angle;
            m_mousex = x;
            m_mousey = y;
            repaint();
            return true;
        }
}
```

The above should be quite easy (if you've ever written an applet). I will not explain the IO nor the threading in this code. The code starts up by getting the map file and loading it. The format of the map file is shown below:

```
1  100 800 100 500 "FFFFFF"
2  200 500 200 400 "FFFF00"
3  400 800 100 800 "FF00FF"
4  400 700 400 800 "00FFFF"
5  500 700 400 700 "FF0000"
6  500 800 500 700 "0000FF"
7  800 800 500 800 "FFFF00"
8  800 500 800 800 "FF00FF"
9  700 500 800 500 "00FFFF"
10  700 400 700 500 "FF00FF"
11  800 400 700 400 "00FF00"
12  800 100 800 400 "FF00FF"
13  500 100 800 100 "00FF00"
14  500 200 500 100 "FFFF00"
15  400 200 500 200 "FF00FF"
16  400 100 400 200 "00FFFF"
17  100 100 400 100 "FF0000"
18  100 400 100 100 "0000FF"
19  200 400 100 400 "00FF00"
20  100 500 200 500 "0000FF"
21  300 400 300 500 "FFFF00"
22  400 400 300 400 "00FFFF"
23  400 300 400 400 "FF00FF"
24  500 300 400 300 "FFFFFF"
25  500 400 500 300 "FFFF00"
26  600 400 500 400 "FF00FF"
27  600 500 600 400 "00FFFF"
28  500 500 600 500 "FFFFFF"
29  500 600 500 500 "FF00FF"
30  400 600 500 600 "FFFFFF"
31  400 500 400 600 "00FF00"
32  300 500 400 500 "FF00FF"
```

With first column being the number of the line (wall), the second column being the `x` coordinate of the starting point of the line. The third column being the `y` coordinate of the starting point of the line. The forth column being the `x` coordinate of the ending point of the line. The fifth column being the `y` coordinate of the ending point of the line. And lastly, the sixth column is the color of the line (wall) in `RGB` format (similar to the way color is represented in `HTML` documents).

Once the map is loaded, we create the tree. Once that's done, we create a double buffer surface to draw into. All that action inside the `init()` method of the applet! After that, we simply fall into the main loop (the `run()` method), and render the tree! The main loop checks to see if there are arrow keys pressed, if they are, then the eye's position is

updated and `redraw()` method is called. The `redraw()` method effectively calls the `update()` method, which clears the double buffer surface, and call's tree's method to render the tree. It then calls the `paint()` method to paint the double buffer surface to the screen area of the applet.

The `renderLine()` method, referred to earlier, is not the best that it could be (it even has lots of bugs!). It is badly written, and is not very efficient. But still, it does a rather satisfactory job at rendering a perspective representation of the line onto the double buffer surface. (besides, this is not even the subject of this section)

You can see the whole thing in action by clicking here!

Well, that's mostly it for Binary Space Partition Trees. What I'd suggest you do is expand on the above applet. Start by improving the `renderLine()` method. Then, try to do a front to back tree traversal instead of the easy back to front traversal approach taken in this tutorial. Anyway, I guess you get the picture: Trees are wonderful data structures, and there are TONS of useful algorithms that use them.

Doing a bit more: This part is added some time after I've initially written this section. Just to show what exactly can be done with VERY minimal effort. You can easily implement lighting! You already have the Z distance of each line, so, all you'll have to do is brighten or darken up the color, and you're done! For example, placing the following lines at the end of `renderLine()` would do the trick:

```
double light = (z1 + z2) / 3;
int R = (R=(int)(light*((l[4] & 0xff0000)>>16))) > 0xFF ? 0xFF:R;
int G = (G=(int)(light*((l[4] & 0x00ff00)>>8))) > 0xFF ? 0xFF:G;
int B = (B=(int)(light*(l[4] & 0x0000ff))) > 0xFF ? 0xFF:B;
double_graphics.setColor(new Color(R,G,B));
```

This might be an over-simplified approach (and will probably suck too much for anything professional), but for our little applet, it's just perfect! Click here to see this new version. I will not include the sources for this modified version in this document, however, they are available inside the ZIP file linked on top.

Kitchen Sink Methods...

In Java, as with everything else in this world, there are additional bells and whistles. They do not restrict us, and offer new ways of doing things. For the purpose of this tutorial, I've named these bells and whistles the *Kitchen Sink Methods* (since they're not directly part of Data Structures, and their inclusion metaphorically adds a kitchen sink to something which has everything *but* the kitchen sink) OK! I admit it, I'm not a creative person when it comes to picking names.

Programs in this section might not be portable nor implemented in the most efficient way. They serve as guides in introducing some of these bells and whistles, which might or might not be useful.

Java Native Interface (JNI)...

I never have

We have all heard about the *Java Native Interface* (JNI), and about a ton of reasons to not use it. The simple truth, however, is that at times, Java is not as fast as we would like it to be. It does not offer system specific features which make other programming languages so powerful.

One might argue that we do not need any system specific features, and could happily exist in the sandbox that the JavaVM provides. In some real world applications, however, performance is a requirement. Imagine writing a full blown game in Java. Without Java3D, you would have quite a headache trying to use your 3D acceleration hardware. Or imagine writing a disk defragmenter in Java, how would you go about doing that?

The answer lies in *Java Native Interface* (JNI). It allows Java application to access methods written in other programming languages (most commonly in C/C++). The process goes like this: we write Java code which defines these `native` methods, use `javac` to compile the java code, then run `javah` on the resulting `class` file. That generates the `.h` include file (for C/C++). We use the method definition from the `.h` file to implement our `native` version of the method. When done, we compile the C/C++ module into a shared library (DLL under Win32, shared lib under UNIX). We then `load` the library from within our Java application, and call the `native` method just as if it were a regular Java method. Sounds interesting, doesn't it?

It gets better. Since all we are using is a shared library, we could theoretically use ANY programming language to write our native code. We could even use Assembler, COBOL, or any other arcane language. We will not do it in this tutorial though, and will go with the plain old C.

Here is where the system specific part of this section creeps in. Under Microsoft Windows, we will use Microsoft Visual C++ v6 Enterprise Edition to compile our C code (any other compiler capable of generating DLL files should do). Under UNIX (or more specifically: SunOS v5.7 running on Spark 5), we will use standard gcc. The generated DLL file, and a Solaris shared lib will be included with the sources archive for this document)

To get started, lets prepare a few things. Under Windows, go to your JDK (or Java SDK as they like to call it now a days) directory and copy all the files from the include directory into your VC++ include directory. Do not forget the file(s) in the <JDK dir>/include/win32 directory; copy all of them to your VC++ include directory. Do the same for all the lib file(s) in <JDK dir>/lib directory; copy those into your VC++ lib directory.

Under Solaris, unless you are the administrator, you will not be able to write to the gcc standard include directory, so, your best bet is to copy all the above mentioned to your project directory (that will make it simpler to point to them). Note that in the JDK 1.2 Solaris version there is no lib file that needs to be linked into your native code.

Once that is done, you are ready to write code! For the purpose of illustrating the uses of JNI, we will convert our quicksort method into native C code. The example is nice enough to illustrate the use of Java arrays in native code, and is practically useful (the C version is faster than the identical Java version).

Note,. that this section will not explain the workings for Quicksort. You are recommended to go and read the Quicksort section of this document before continuing. Also realize the the things described in this particular section are not hard; they're quite easy. As soon as you successfully call your first `native` method, everything will be clear.

Continuing from that encouraging sentence, lets get to writing the code! We will start by implementing our main application code (which will change later). So far, all we need is this:

```
import java.lang.*;
import java.io.*;

public class pQuicksortNative{

    public static native void qsort(int[] c);

    public static void main(String[] args){
        int i;
        int[] arr = new int[20];
        System.out.println("inserting: ");
        for(i=0;i<arr.length;i++){
            arr[i] = (int)(Math.random()*99);
            System.out.print(arr[i]+" ");
        }
        qsort(arr);
        System.out.println("\nsorted: ");
        for(i=0;i<arr.length;i++)
            System.out.print(arr[i]+" ");
        System.out.println("\nDone ;-)");
    }
}
```

This code looks nearly identical to the testing code in Quicksort section. A thing you should definitely notice is the absence of actual implementation of `qsort()` method. The declaration of `qsort()` contains a keyword `native`, which tells the Java compiler that the implementation will be loaded as a library at runtime. Implementation can be written in any language, as long as it is a library load-able at runtime. (I'm sure there are many ways of bending this definition, but that's what is *generally* assumed by the `native` declaration.)

After compiling the above code, run `javah` on the resulting `class` file.

```
> javac pQuicksortNative.java
> javah pQuicksortNative
```

This should generate a file named `pQuicksortNative.h`, which is your C/C++ include file. It contains the declarations for all the `native` methods of a given class. In our simple example, the include file should look something like:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class pQuicksortNative */

#ifndef _Included_pQuicksortNative
#define _Included_pQuicksortNative
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     pQuicksortNative
 * Method:    qsort
 * Signature: ([I)V
 */
JNIEXPORT void JNICALL Java_pQuicksortNative_qsort
  (JNIEnv *, jclass, jintArray);

#ifdef __cplusplus
}
```

```
#endif
#endif
```

The scary message on top doesn't mean much (of course you can edit the file), but it makes very little point to do so. This generated file can be used with either C or C++ code. Our example will use plain C, but the few key differences between C++ will be pointed out.

All we need to do now, is write our C module containing the implementation of `Java_pQuicksortNative_qsort`. To make the process simple, we will simply cut and paste the declaration into a new file, and continue from there. We then take the Java source which we did in our Quicksort section, convert it to C, and that's more or less the whole job. The C module (named `qsort.c`) follows.

```
#include "pQuicksortNative.h"

JNIEXPORT void JNICALL
Java_pQuicksortNative_qsort(JNIEnv * jniEnv,
    jclass javaClass,jintArray arr){

    int i,j,left = 0,right,stack_pointer = -1;
    int stack[128];
    int swap,temp;

    /* get actual array & it's size */
    jint* c = (*jniEnv)->GetIntArrayElements(jniEnv,arr,0);
    right = (*jniEnv)->GetArrayLength(jniEnv,arr) - 1;

    for(;;){
        /* see if to do insertion sort or quicksort */
        if(right - left <= 7){
            /* simple insertion sort */
            for(j=left+1;j<=right;j++){
                swap = c[j];
                i = j-1;
                while(i>=left && c[i] > swap)
                    c[i+1] = c[i--];
                c[i+1] = swap;
            }
            if(stack_pointer == -1)
                break;
            right = stack[stack_pointer--];
            left = stack[stack_pointer--];
        }else{
            /* quicksort */

            /* find the median */
            int median = (left + right) >> 1;
            i = left + 1;
            j = right;

            /* swap the median */
            c[median] ^= c[i];
            c[i] ^= c[median];
            c[median] ^= c[i];

            /* make sure: c[left] <= c[left+1] <= c[right] */
            if(c[left] > c[right]){
                c[left] ^= c[right];
                c[right] ^= c[left];
                c[left] ^= c[right];
            }if(c[i] > c[right]){
                c[i] ^= c[right];
                c[right] ^= c[i];
                c[i] ^= c[right];
            }if(c[left] > c[i]){
                c[i] ^= c[left];
                c[left] ^= c[i];
                c[i] ^= c[left];
            }
            temp = c[i];
            for(;;){
                do i++; while(c[i] < temp);
                do j--; while(c[j] > temp);
                if(j < i)
                    break;
                c[i] ^= c[j];
                c[j] ^= c[i];
                c[i] ^= c[j];
            }
```

```
            c[left + 1] = c[j];
            c[j] = temp;
            if(right-i+1 >= j-left){
                stack[++stack_pointer] = i;
                stack[++stack_pointer] = right;
                right = j-1;
            }else{
                stack[++stack_pointer] = left;
                stack[++stack_pointer] = j-1;
                left = i;
            }
        }
    }
    /* release array */
    (*jniEnv)->ReleaseIntArrayElements(jniEnv,arr,c,0);
}
```

The code changed only slightly from it's Java implementation. One thing that should strike you as strange is the use of function pointers in structures (the `GetArrayLength()`, etc.). It is correct to assume that in C++, this code becomes a bit simpler. And something like:

```
(*jniEnv)->ReleaseIntArrayElements(jniEnv,arr,c,0);
```

Will reduce to something like the following in C++:

```
jniEnv->ReleaseIntArrayElements(arr,c,0);
```

We're not in C++ however. Don't think that the C++ way is faster or more efficient however; it does exactly the same thing as the C method, and thus, takes exactly the same time to execute.

Now, what are those weird functions which we call? Why do we need that `GetIntArrayElements()` or `GetArrayLength()`, and why do we need to call `ReleaseIntArrayElements()` when we are finished sorting? Good question; the answer is in the way JavaVM works.

In C, we have the good old `malloc()` & `free()` to allocate/free memory. In C++, we have the all versatile `new` & `delete` operators. In Java, we have the `new` operator and the good old garbage collector. This fact that memory in Java is not fully controlled by the programmer, but by the Virtual Machine makes for some interesting issues when letting C code play around with memory managed by the JavaVM.

The `native` module expects memory to stay in one place. It does not want things to be wiped out or garbage collected when it is using them. The primary reason for calling `GetIntArrayElements()` method is to notify the JavaVM that we want to use that block of memory. The JavaVM has several options at this point. It can pin-down this block of memory (prevent it from being moved), or it can create a copy of the original data, and let us play around with the copy. No matter what it does, when we are done using the memory, we have to notify the JavaVM that we are done using it (so that it can unpin the memory, or copy the new memory block over the old one).

It is interesting to note that under Microsoft Windows it primarily seems to give you the pinned down memory, while under Solaris, it tends to give you a copy to work with. You can tell whether it's a copy or not by passing the address of a `jboolean` variable as the fourth parameter to `GetIntArrayElements()`. I suggest you look though the JDK and JavaVM documentation for a more detailed explanation.

If you are really into it, you might have noticed other changes in the code. Some of the swap code now uses `XOR` instead of a temporary variable to swap numbers. I think it's kind of cute. (but only works for numbers, not objects)

Gotten this far, it would be a shame not to compile it! Using command line options to generate a `DLL` under Microsoft Windows:

```
> cl /GD /LD qsort.c
```

and under UNIX (Solaris):

```
> gcc -shared -I $HOME/javadata -o qsort.a qsort.c
```

Assuming you are in the correct directory, your `PATH` is setup correctly, and everything else works, you should have a `qsort.dll` under Windows, and/or a `qsort.a` shared lib under UNIX.

Now that you got that done, you can go ahead and modify your Java application to load the library. The code for a contemporary `load()` procedure would look something like this:

```
static{
    System.load("c:/projects/javadata/qsort.dll");
}
```

Of course, the absolute path to the library will be different. This `System.load()` method requires that we pass the complete full path of the shared library to it. There is another method: `System.loadLibrary()` that only requires the

name of the library. This `loadLibrary()` assumes that the library is inside some system directory, other than that, the idea is the same. And now, for a paragraph of preferences:

I found it simpler to use the `System.load()` as opposed to `System.loadLibrary()`. In the former one, you specify the full path, and you're done with. With `System.loadLibrary()` however, it's not that simple. For one, you have to specify just the name of the `DLL` under Windows, without the actual `.DLL` extension. This technique obviously doesn't work under UNIX, where there are no `DLL` files. Under both systems, the library loaded by `System.loadLibrary()` has to be inside some system library directory. Under Microsoft Windows, it is supposedly the `\Windows\System`, and under UNIX, it is supposedly the `/lib` (among other things). Everything will still work under Windows if the `DLL` is not in the system directory, but not under UNIX. One could argue that you can modify the `Properties` directly from within the Java application, and make it think that the current directory is a system's lib directory, but that's a pain in the neck. Because of these reasons, we will use `System.load()` throughout this document.

Modifying the original Java test application gives us:

```
import java.lang.*;
import java.io.*;

public class pQuicksortNative{

    static{
        System.load("c:/projects/javadata/qsort.dll");
    }

    public static native void qsort(int[] c);

    public static void main(String[] args){
        int i;
        int[] arr = new int[20];
        System.out.println("inserting: ");
        for(i=0;i<arr.length;i++){
            arr[i] = (int)(Math.random()*99);
            System.out.print(arr[i]+" ");
        }
        qsort(arr);
        System.out.println("\nsorted: ");
        for(i=0;i<arr.length;i++)
            System.out.print(arr[i]+" ");
        System.out.println("\nDone ;-)");
    }
}
```

Not much change from the original one, huh? This is obviously a Microsoft Windows version (so much for code portability...) We can just replace that absolute path with a UNIX path to that of UNIX lib, and it will work there as well. To make it more portable, we will move the library loading code into the `main()` method, and will accept the path to the lib from command line. So, under UNIX, you'll pass the path to UNIX lib, and under Windows, you'll pass the path to that `DLL`. For example:

```
import java.lang.*;
import java.io.*;

public class pQuicksortNative{

    public static native void qsort(int[] c);

    public static void main(String[] args){

        if(args.length > 0)
            try{
                System.load(args[0]);
            }catch(java.lang.UnsatisfiedLinkError e){
                System.out.println("bad lib name: "+args[0]);
                return;
            }
        else{
            System.out.println("include lib name as parameter");
            return;
        }

        int i;
        int[] arr = new int[20];
        System.out.println("inserting: ");
        for(i=0;i<arr.length;i++){
            arr[i] = (int)(Math.random()*99);
            System.out.print(arr[i]+" ");
        }
```

```
        qsort(arr);
        System.out.println("\nsorted: ");
        for(i=0;i<arr.length;i++)
            System.out.print(arr[i]+" ");
        System.out.println("\nDone ;-)");
    }
}
```

The output of such a program follows:

```
C:\projects\javadata>java pQuicksortNative c:\projects\javadata\qsort.dll
inserting:
55 3 46 54 18 89 0 71 89 45 31 81 67 76 57 4 97 48 59 60
sorted:
0 3 4 18 31 45 46 48 54 55 57 59 60 67 71 76 81 89 89 97
Done ;-)
```

Alternatively, you might have a configuration file which stores this kind of information (what system it is running on, which lib files to load, etc.) An ugly thing to do could be to name all libraries the same name, so that the code always knows what to load. This situation can lead to wrong libraries being loaded, and other horrible things.

I am guessing that's enough of drilling this stuff into your mind. If you want to learn more about `native` methods, you can pick up a lot from the Java documentation found at Sun's web-site. If you don't want to learn more, then what you've learned so far, should be more than enough for the purposes of general knowledge.

Bibliography...

I have gotten enough responses from people asking me to list good references for all this info, so, here I am, trying to list everything that is relevant. It is by no means a complete list, but it should provide the reader with adequate reference. I will supply the name, author(s), and a very brief opinionated description. Most book stores will let you search for the title, so, I don't think ISBN is relevant in most cases.

Most relevant:

Introduction to Algorithms, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Riverst. It is a rather large book, which not only covers all the common data structures, but most common algorithms as well. A really nice book to have as a reference.

The Art of Computer Programming Volume 1, Fundamental Algorithms, Donald E. Knuth. This is *the* reference for all your computer programming needs. I suggest getting all 3 volumes, in addition to most of Knuth's publications.

Data Structures Using C and C++, Second Edition, by Yedidyah Langsam, Moshe J. Augenstain, Aaron M. Tenenbaum. A very nice book, covering most things I could think of about data structures. The code is not very readable though, and could have been a bit clearer. Some concepts lack concrete working examples. Incidentally, I had most of these authors as my professors for some class or another.

The Art of Computer Programming Volume 3, Sorting and Searching, Second Edition, by Donald E. Knuth. Without an argument, this is the BEST reference to get on sortint and searching. I don't think I used anything out of other volumes of the series, but ALL 3 volumes are definitely worth getting. It's a classic!

Computer Algorithms, Introduction to Design and Analysis, Second Edition, by Sara Baase. A nice book with lots of information on various algorithms. I still think The Art of Computer Programming by Donald Knuth is much better.

Java Distributed Objects, The Authoritative Solution by Bill McCarty and Luke Cassady-Dorion. This book is the ultimate book on OOP concepts and distributed systems using Java. It covers CORBA, RMI, DCOM, etc. It even implements the same program using ALL those approaches, including the plain sockets implementations!

Database System Concepts by Abraham Silberschatz, Henry F. Korth, S. Sudarshan. The usual database book. You'll need one of these, since in the real world, most companies require a whole lot of DB experience.

C/C++ Annotated Archives By Art Friedman, Lars Klander, and Mark Michaelis. This book is kind of a source book. It has source for binary trees and other interesting stuff.

Discrete Mathematics, Forth Edition, by Richard Johnsonbaugh. A very readable book on math as it relates to computers. Covers some typical data structures, including trees, while illustrating some interesting algorithms that use them. I like this book mostly for it's simple illustration of some algorithms. If all books seem too high level, this is the one to get.

Data Structures and Algorithms in Java Michael T. Goodrich, Roberto Tamassia, 1998. "The text focuses on applications, with numerous Java code examples and object-oriented software design patterns. Animations illustrate data structures and algorithms in a clear visual manner without the need for lenghty mathematical derivations. Website devoted to the book: http://www.wiley.com/college/cs2java."

Less relevant:

Borland C++ v4.5 Object-Oriented Programming, Forth Edition, by Ted Faison. Don't get me wrong, I know that this book is outdated, and that nobody in their right mind would use Borland v4.5 now, however, this book is in my opinion the best Object Oriented Programming book I've seen. It covers C++ and most related topics. This is not how to use Borland C++ type of a book, it's really full of Object Oriented theory and pure C++ examples to back it up. I'd personally look for future editions by the same author.

Artificial Intelligence, Structures and Strategies for Complex Problem Solving, Second Edition, by George F. Luger, William A. Stubblefield. Most of this book is fairly high level stuff, but it does show some interesting ways to play around with tree based structures, and how they relate to games. It has lots of different methods for tree traversal.

Practical UNIX Programming, A Guide to Concurrency, Communication, and Multithreading, by Kay A. Robbins, Steven Robbins. This is the book to get for system programming. The way in which it relates to this document is that it has lots of multithreading stuff, it describes it inside out. It's also a pretty good reference on basic networking. (well, maybe this document didn't have all that, but it's still a good reference...)

Java 1.1, Developer's Guide, Second Edition, by Jamie Jaworski. This is my second, and probably the last Java book. It's a good reference for all those classes, etc., but it really has nothing new for somebody who already knows Java. No interesting algorithms, just a systematic coverage of the language, and it's supporting API. (hey, I needed to put in at least one Java reference into this list ;-)

Code Complete, by Steve McConnell. This books talks about code design. How to structure code, how to plan, layout, write, test, and intergrate your code. It is definitely a "*Practical Handbook of Software Construction.*"

Irrelevant, but could be interesting (graphics):

Gardens of Imagination by Christopher Lampton This book is pretty good for a total beginner in graphics. It is a bit dated though, since it only has DOS code, which most of the time, simply doesn't run under current operating systems. BEGINNER LEVEL

Tricks of the Game Programming Gurus, by LaMothe, Ratcliff, Seminatore & Tyler Best Startup book I've seen. It is interesting to read, and offers quite a bit of inspiration. However, practically speaking, this book is outdated (some chapters could be interesting, but...) BEGINNER LEVEL

3D Game Programming With C++ by John De Goes That's the book that taught me how to create a window, under Windows, set palette, etc., however, other than that, it has nothing new. It does cover quite a bit of material, but for some strange reason, I didn't find it useful. You do need to know C++ before getting this book though. LOWER INTERMEDIATE LEVEL.

Computer Graphics, Principles and Practice, Second Edition in C, Foley, van Dam, Feiner, Hughes. This is the definitive book on Computer graphics. If you need a book that has everything, then get this one!

Image Processing In Java by Douglas A. Lyon. Includes lots of cool algorithms for all kinds of fun image effects. Very readable and easy to follow. (I actually read the whole thing in an evening.)

Algorithmic Geometry by J-D Boissonnat and M. Yvinec. A very technical book on algorithms like generating convex hulls, triangulating, and other fun things. Note that this is not a general type of Graphics book, it's a very technical math-like book.

Matrix Computations, Third Edition, by Gene H. Golub and Charles F. Van Loan. Another very techy book. Anything you ever wanted to know about Matrix manipulation on a computer. It tries to be a programming book, but in my eyes, it's a math book.

Computer Graphics by Roy A. Plastock and Gordon Kalley. A Schaum's Outline series book. Covers all the required graphics concepts. A bit brief for my taste though. The book is very cheap though (23 Canadian Dollars), so, if you're on a tight budged, this might be the one to get. I always tend to get techy Schaum's Outlines, they're cheap, concise, and attempt cover the subject more or less completely.

ZEN of Gaphics Programming, by Michael Abrash Very interesting, and the most useful book I read at the time (when it came out). It has code, and is quite inspirational. HIGHER INTERMEDIATE LEVEL

Michael Abrash's Graphics Programming Black Book Special Edition This book contains full text of several other books, including most of the text from ZEN of Graphics Programming. It is a bit more interesting book. It's final chapters talk quite a bit about the theory behind Quake engine & BSP trees. HIGHER INTERMEDIATE LEVEL

Computer Graphics, C version, Second Edition, by Donald Hearn, M. Pauline Baker One of those text-book type books. It covers everything you could possibly think up, but at times is quite brief. It's a good reference to have though (in case you forget how to do gouraud shading or something). ADVANCED LEVEL

OpenGL Programming Guide, Third Edition, The Official Guide To Learning OpenGL. Best OpenGL reference I've seen. Has most of the stuff anybody would need. Nothing system specific though, which is good.

Digital Typography, by Donald E. Knuth. A nice introduction and description of the field of making pretty text. Briefly describes TeX, Metafont, and other Knuth's creations, along with a few algorithms on aligning text, and other fun stuff found in TeX.

Pre-calculus Mathematics, 3rd Edition, by Hungerford, Mercer Most of the problems that come up in graphics are mathematical ones. This book might seem like a joke to some math major, but that's the book which has most of the equations for graphics programming. (like finding the determinant of an NxN matrix ;-) BEGINNER LEVEL

Totally Irrelevant, but Might be Interesting (theory):

Introduction to Languages and the Theory of Computation, Second Edition, by John C. Martin. This book goes through quite a bit of theory behind scanners, parsers, and other interesting topics. This is the type of book which programmers wanting to design languages and write compilers use. ADVANCED LEVEL

Crafting a Compiler with C, by Charles N. Fischer, Richard J. LeBlanc Jr. From this book you'll learn the basics of compiler design from the ground up. No previous knowledge necessary. However, I'd recommend reading the above book before, since that will make it a LOT easier for you to comprehend some of the ideas. Note that it has a few bugs in the LR parsing example, nothing wrong with algorithms as far as I know. ADVANCED LEVEL

Advanced Compiler Design and Implementation, by Steven S. Muchnick. A lot more detailed & more practical book than the above one. However, that *advanced* word in the title is not a joke. The book does jump right into things like optimization, without fully covering compiler basics. Basically, if you've written a compiler and want to make it run faster, or port it, etc., that's the book to get. REALLY ADVANCED LEVEL

Lex & Yacc, by John R. Levine, Tony Mason, and Doug Brown. This O'Reilly's reference is one of the best descriptions (with examples) of Lex and Yacc I have yet to come across.

Programming Languages, Concepts & Constructs, by Ravi Sethi. This book is too simple for anybody who read at least one of the books mentioned above. It could be interesting as a history book; like which computer language evolved out of which, and the general historical significance of events which led to the development of particular languages, etc.

The Data Compression Book, Second Edition, by Mark Nelson and Jean-Loup Gailly. A rather nice, slow paced introduction to data compression using all kinds of algorithms.

Numerical Recipes in C, The Art of Scientific Computing, Second Edition, by William H. Press, William T. Vetterling, Saul A. Teukolsky, Brian P. Flannery. This book, along with **The Art of Computer Programming Volume 2, by Donald E. Knuth.** has just about every single numerical algorithm most people can think of. For a *very simple* introduction to cryptography, try getting **Applied Cryptography by Bruce Schneier**. Note that some people claim Numerical Recipes has a few bugs in it; see http://math.jpl.nasa.gov/nr/nr.html for more information. ADVANCED LEVEL

An Introduction to The Theory of Numbers, Fifth Edition, by Ivan Niven, Herbert S. Zuckerman, Hugh L. Montgomery. This book has everything you'll ever need to know about numbers. So far, this book has the best description of RSA I've ever seen. Very technical, but that's exactly what makes this book good.

Simulation, Second Edition, by Sheldon M. Ross. A general book on simulations. Covers algorithms on generating random numbers, sampling various distribution functions, etc.

Irrelevant (General):

Data And Computer Communications, Fifth Edition, by William Stallings. A nice book on general Networking concepts. Very technical when it comes to hardware etc., but still fun. Gives the basics of lots of network protocols, including a fairly good description of SMTP.

Operating System Concepts, Fifth Edition, by Abraham Silberschatz, and Peter Baer Galvin. General purpose Operating Systems book. Has most of the relevant algorithms, etc. Offers nice descriptions of various actual existing operating systems, including UNIX, Linux, and Windows NT.

UNIX System Administration Handbook, Second Edition, by Evi Nemeth, Garth Snyder, Scott Seebass, Trent R. Hein. An indispensable book on System Administration; best that I've seen!

UNIX System V, A practical Guide, Third Edition, by Mark G. Sobell. A fairly descriptive book of the basics of UNIX. Simple and easy to follow, with nice examples of most of the interesting stuff. If you want to learn shell programming, this is the book to get.

Mastering Perl 5, by Eric C. Herrmann. A complete reference for Perl. I think that said it all.

COBOL, From Micro to Mainframe, Third Edition, by Robert T. Grauer, Carol Vazquez Villar, Arthur R. Buss. Nicely covers this arcane language, with more or less real world examples. Lots of code, lots of pages, but COBOL always tended to be wordy.

Visual Basic 5, by Alan Eliason and Ryan Malarkey. Since I put a COBOL book on this list, it's only fair that I put a Visual Basic book as well.

Programming and Customizing the PIC Microcontroller by Mike Predko. A simple description of what is involved in programming and experimenting with microcontrollers. Truthfully, when it came to actually writing code for one of these, the PIC data sheet proved a lot more useful than this book. This book is still a nice kick-start if you really have no idea what's going on.

This is just about one shelf worth of books (I'm not gonna go through another one...) One thing I would like to mention is that you should always also get a system programming book as well. For example, if you are doing programming under WindowsNT, you should get something like *Windows NT4 Programming from the Ground Up by Herbert Schildt*. Or something similar. It does help to know how the underlying operating system works.

Special Thanks...

Due to the enormous amounts of corrections I get from people, I'd like to thank those that make significant contributions. If your name is not on this list (and you believe it should be), just e-mail me, and I'll put it there. Anyway, special thanks goes to:

CheezHankrn for some constructive criticism.
Shalva S. Landy for lots of spelling/grammar corrections.
Oliver Neuberger for fixing the `postrav()` method.
Dan Perl for noticing some remnants of the postrav() problem.
Roosevelt Victor for thoroughly testing a lot of this code.
Andrey Salaev for fixing the peek() method in pLinkedList (no one has apparently noticed the bug for many long years...)
Bjørn Harald Olsen for noticing some more remnants of the postrav() problem.

and many others...

Contact Info...

The only sure way to find me, is to use my e-mail @theparticle.com. Yep, it's my domain which will hopefully stay with me.

I encourage feedback!

Particle
particle at NO SPAM the particle dot com
http://www.theparticle.com/

Copyright © 1996-2001, Particle

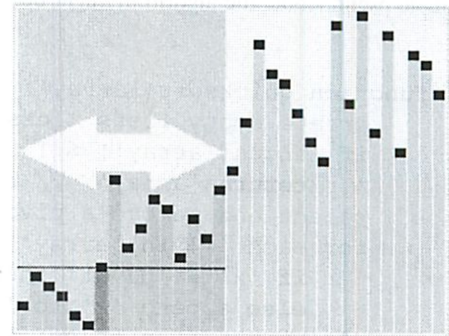
Quicksort

From Wikipedia, the free encyclopedia

Quicksort is a sorting algorithm developed by Tony Hoare that, on average, makes $O(n\log n)$ (big O notation) comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons, though this behavior is rare. Quicksort is often faster in practice than other $O(n\log n)$ algorithms.[1] Additionally, quicksort's sequential and localized memory references work well with a cache. Quicksort can be implemented as an in-place sort, requiring only $O(\log n)$ additional space.[2]

Quicksort (also known as "partition-exchange sort") is a comparison sort and, in space efficient implementations, is not a stable sort.

Quicksort



Visualization of the quicksort algorithm. The horizontal lines are pivot values.

Class	Sorting algorithm
Worst case performance	$O(n^2)$
Best case performance	$O(n \log n)$
Average case performance	$O(n \log n)$
Worst case space complexity	$O(n)$

Contents

- 1 History
- 2 Algorithm
  - 2.1 Simple version
  - 2.2 In-place version
  - 2.3 Implementation issues
    - 2.3.1 Choice of pivot
    - 2.3.2 Optimizations
    - 2.3.3 Parallelization
- 3 Formal analysis
  - 3.1 Randomized quicksort expected complexity
  - 3.2 Average complexity
  - 3.3 Space complexity
- 4 Selection-based pivoting
- 5 Variants
- 6 Comparison with other sorting algorithms
- 7 See also
- 8 Notes
- 9 References
- 10 External links

History

The quicksort algorithm was developed in 1960 by Tony Hoare while in the Soviet Union, as a visiting student at Moscow State University. At that time, Hoare worked in a project on machine translation for the National Physical Laboratory. He developed the algorithm in order to sort the words to be translated, to make them more easily matched to an already-sorted Russian-to-English dictionary that was stored on magnetic tape.[3]

Thaw

Algorithm

Quicksort is a divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.

The steps are:

1. Pick an element, called a *pivot*, from the list.
2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

The base case of the recursion are lists of size zero or one, which never need to be sorted.

Simple version

In simple pseudocode, the algorithm might be expressed as this:

```
function quicksort('array')
    create empty lists 'less' and 'greater'
    if length('array') ≤ 1
        return 'array'  // an array of zero or one elements is already sorted
    select and remove a pivot value 'pivot' from 'array'
    for each 'x' in 'array'
        if 'x' ≤ 'pivot' then append 'x' to 'less'
        else append 'x' to 'greater'
    return concatenate(quicksort('less'), 'pivot', quicksort('greater'))
```

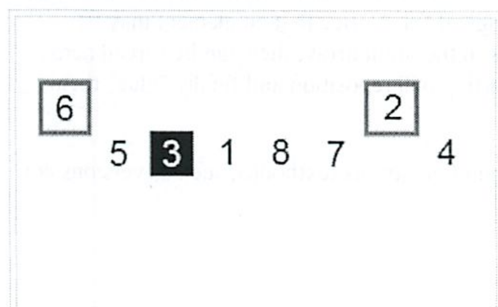
Need to learn all of them + differences

Notice that we only examine elements by comparing them to other elements. This makes quicksort a comparison sort. This version is also a stable sort (assuming that the "for each" method retrieves elements in original order, and the pivot selected is the last among those of equal value).

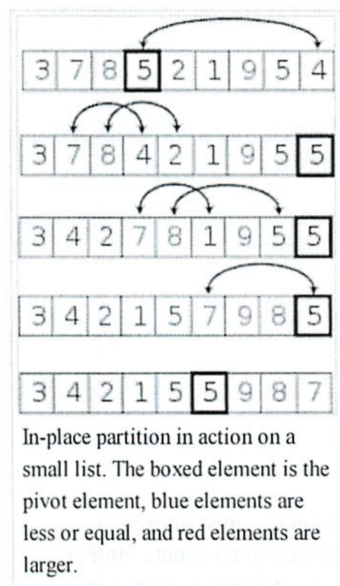
The correctness of the partition algorithm is based on the following two arguments:

- At each iteration, all the elements processed so far are in the desired position: before the pivot if less than the pivot's value, after the pivot if greater than the pivot's value (loop invariant).
- Each iteration leaves one fewer element to be processed (loop variant).

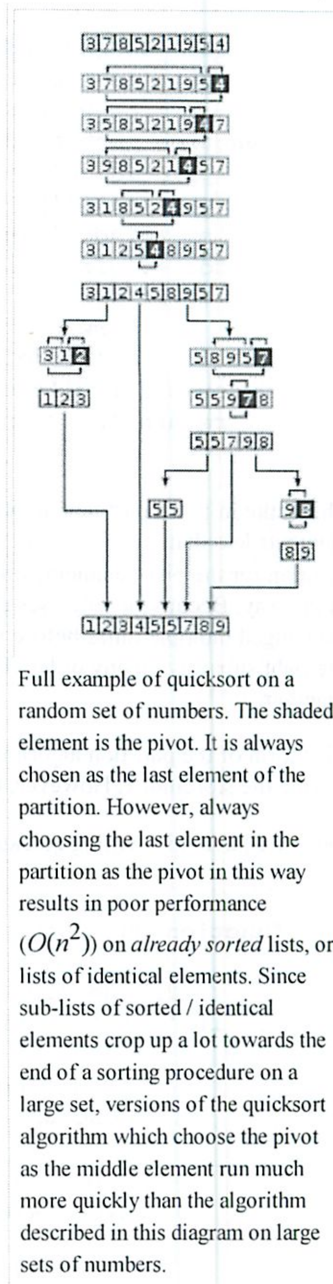
The correctness of the overall algorithm can be proven via induction: for zero or one element, the algorithm leaves the data unchanged; for a larger data set it produces the concatenation of two parts, elements less than the pivot and elements greater than it, themselves sorted by the recursive hypothesis.



An example of quicksort.



In-place partition in action on a small list. The boxed element is the pivot element, blue elements are less or equal, and red elements are larger.



Full example of quicksort on a random set of numbers. The shaded element is the pivot. It is always chosen as the last element of the partition. However, always choosing the last element in the partition as the pivot in this way results in poor performance ($O(n^2)$) on *already sorted* lists, or lists of identical elements. Since sub-lists of sorted / identical elements crop up a lot towards the end of a sorting procedure on a large set, versions of the quicksort algorithm which choose the pivot as the middle element run much more quickly than the algorithm described in this diagram on large sets of numbers.

In-place version

The disadvantage of the simple version above is that it requires O(n) extra storage space, which is as bad as merge sort. The additional memory allocations required can also drastically impact speed and cache performance in practical implementations. There is a more complex version which uses an in-place partition algorithm and can achieve the complete sort using O(log n) space (not counting the input) use on average (for the call stack):

```
// left is the index of the leftmost element of the array
// right is the index of the rightmost element of the array (inclusive)
//   number of elements in subarray = right-left+1
function partition(array, 'left', 'right', 'pivotIndex')
   'pivotValue' := array['pivotIndex']
   swap array['pivotIndex'] and array['right']  // Move pivot to end
   'storeIndex' := 'left'
   for 'i' from 'left' to 'right' - 1  // left ≤ i < right
       if array['i'] < 'pivotValue'
           swap array['i'] and array['storeIndex']
           'storeIndex' := 'storeIndex' + 1
   swap array['storeIndex'] and array['right']  // Move pivot to its final place
   return 'storeIndex'
```

This is the in-place partition algorithm. It partitions the portion of the array between indexes *left* and *right*, inclusively, by moving all elements less than `array[pivotIndex]` before the pivot, and the equal or greater elements after it. In the process it also finds the final position for the pivot element, which it returns. It temporarily moves the pivot element to the end of the subarray, so that it doesn't get in the way. Because it only uses exchanges, the final list has the same elements as the original list. Notice that an element may be exchanged multiple times before reaching its final place. Also, in case of pivot duplicates in the input array, they can be spread across the right subarray, in any order. This doesn't represent a partitioning failure, as further sorting will reposition and finally "glue" them together.

This form of the partition algorithm is not the original form; multiple variations can be found in various textbooks, such as versions not having the storeIndex. However, this form is probably the easiest to understand.

Once we have this, writing quicksort itself is easy:

```
function quicksort(array, 'left', 'right')

    // If the list has 2 or more items
    if 'left' < 'right'

        // See "Choice of pivot" section below for possible choices
        choose any 'pivotIndex' such that 'left' ≤ 'pivotIndex' ≤ 'right'

        // Get lists of bigger and smaller items and final position of pivot
        'pivotNewIndex' := partition(array, 'left', 'right', 'pivotIndex')

        // Recursively sort elements smaller than the pivot
        quicksort(array, 'left', 'pivotNewIndex' - 1)

        // Recursively sort elements at least as big as the pivot
        quicksort(array, 'pivotNewIndex' + 1, 'right')
```

Each recursive call to this *quicksort* function reduces the size of the array being sorted by at least one element, since in each invocation the element at *pivotNewIndex* is placed in its final position. Therefore, this algorithm is guaranteed to terminate after at most *n* recursive calls. However, since *partition* reorders elements within a partition, this version of quicksort is not a stable sort.

Implementation issues

Choice of pivot

In very early versions of quicksort, the leftmost element of the partition would often be chosen as the pivot element. Unfortunately, this causes worst-case behavior on already sorted arrays, which is a rather common use-case. The problem was easily solved by choosing either a random index for the pivot, choosing the middle index of the partition or (especially for longer partitions) choosing the median of the first, middle and last element of the partition for the pivot (as recommended by R. Sedgewick).[4][5]

Selecting a pivot element is also complicated by the existence of integer overflow. If the boundary indices of the subarray being sorted are sufficiently large, the naive expression for the middle index, *(left + right)/2*, will cause overflow and provide an invalid pivot index. This can be overcome by using, for example, *left + (right-left)/2* to index the middle element, at the cost of more complex arithmetic. Similar issues arise in some other methods of selecting the pivot element.

Optimizations

Two other important optimizations, also suggested by R. Sedgewick, as commonly acknowledged, and widely used in practice are:[6][7][8]

- To make sure at most O(log N) space is used, recurse first into the smaller half of the array, and use a tail call to recurse into the other.
- Use insertion sort, which has a smaller constant factor and is thus faster on small arrays, for invocations on such small arrays (i.e. where the length is less than a threshold *t* determined experimentally). This can be implemented by leaving such arrays unsorted and running a single insertion sort pass at the end, because insertion sort handles nearly sorted arrays efficiently. A separate insertion sort of each small segment as they are identified adds the overhead of starting and stopping many small sorts, but avoids wasting effort comparing keys across the many segment boundaries, which keys will be in order due to the workings of the quicksort process. It also improves the cache use.

Parallelization

Like merge sort, quicksort can also be parallelized due to its divide-and-conquer nature. Individual in-place partition operations are difficult to parallelize, but once divided, different sections of the list can be sorted in parallel. The following is a straightforward approach: If we have p processors, we can divide a list of n elements into p sublists in $O(n)$ average time, then sort each of these in $O\left(\frac{n}{p}\log\frac{n}{p}\right)$ average time. Ignoring the $O(n)$ preprocessing and merge times, this is linear speedup. If the split is blind, ignoring the values, the merge naively costs $O(n)$. If the split partitions based on a succession of pivots, it is tricky to parallelize and naively costs $O(n)$. Given $O(\log n)$ or more processors, only $O(n)$ time is required overall, whereas an approach with linear speedup would achieve $O(\log n)$ time for overall.

One advantage of this simple parallel quicksort over other parallel sort algorithms is that no synchronization is required, but the disadvantage is that sorting is still $O(n)$ and only a sublinear speedup of $O(\log n)$ is achieved. A new thread is started as soon as a sublist is available for it to work on and it does not communicate with other threads. When all threads complete, the sort is done.

Other more sophisticated parallel sorting algorithms can achieve even better time bounds.[9] For example, in 1991 David Powers described a parallelized quicksort (and a related radix sort) that can operate in $O(\log n)$ time on a CRCW PRAM with n processors by performing partitioning implicitly.[10]

Formal analysis

From the initial description it's not obvious that quicksort takes $O(n \log n)$ time on average. It's not hard to see that the partition operation, which simply loops over the elements of the array once, uses $O(n)$ time. In versions that perform concatenation, this operation is also $O(n)$.

In the best case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size. Consequently, we can make only $\log n$ nested calls before we reach a list of size 1. This means that the depth of the call tree is $\log n$. But no two calls at the same level of the call tree process the same part of the original list; thus, each level of calls needs only $O(n)$ time all together (each call has some constant overhead, but since there are only $O(n)$ calls at each level, this is subsumed in the $O(n)$ factor). The result is that the algorithm uses only $O(n \log n)$ time.

An alternative approach is to set up a recurrence relation for the $T(n)$ factor, the time needed to sort a list of size n. Because a single quicksort call involves $O(n)$ factor work plus two recursive calls on lists of size $n/2$ in the best case, the relation would be:

$$T(n) = O(n) + 2T\left(\frac{n}{2}\right).$$

The master theorem tells us that $T(n) = O(n \log n)$.

In fact, it's not necessary to divide the list this precisely; even if each pivot splits the elements with 99% on one side and 1% on the other (or any other fixed fraction), the call depth is still limited to $100 \log n$, so the total running time is still $O(n \log n)$.

In the worst case, however, the two sublists have size 1 and $n-1$ (for example, if the array consists of the same element by value), and the call tree becomes a linear chain of n nested calls. The ith call does $O(n-i)$ work, and $\sum_{i=0}^{n}(n-i) = O(n^2)$. The recurrence relation is:

$$T(n) = O(n) + T(0) + T(n-1) = O(n) + T(n-1).$$

This is the same relation as for insertion sort and selection sort, and it solves to $T(n) = O(n^2)$. Given knowledge of which comparisons are performed by the sort, there are adaptive algorithms that are effective at generating worst-case input for quicksort on-the-fly, regardless of the pivot selection strategy.[11]

Randomized quicksort expected complexity

Randomized quicksort has the desirable property that, for any input, it requires only $O(n \log n)$ expected time (averaged over all choices of pivots). But what makes random pivots a good choice?

Suppose we sort the list and then divide it into four parts. The two parts in the middle will contain the best pivots; each of them is larger than at least 25% of the elements and smaller than at least 25% of the elements. If we could consistently choose an element from these two middle parts, we would only have to split the list at most $2\log_2 n$ times before reaching lists of size 1, yielding an $O(n \log n)$ algorithm.

A random choice will only choose from these middle parts half the time. However, this is good enough. Imagine that you are flipping a coin over and over until you get k heads. Although this could take a long time, on average only $2k$ flips are required, and the chance that you won't get k heads after $100k$ flips is highly improbable. By the same argument, quicksort's recursion will terminate on average at a call depth of only $2(2\log_2 n)$. But if its average call depth is $O(\log n)$, and each level of the call tree processes at most n elements, the total amount of work done on average is the product, $O(n \log n)$. Note that the algorithm does not have to verify that the pivot is in the middle half - if we hit it any constant fraction of the times, that is enough for the desired complexity.

The outline of a formal proof of the $O(n \log n)$ expected time complexity follows. Assume that there are no duplicates as duplicates could be handled with linear time pre- and post-processing, or considered cases easier than the analyzed. Choosing a pivot, uniformly at random from 0 to n-1, is then equivalent to choosing the size of one particular partition, uniformly at random from 0 to n-1. With this observation, the continuation of the proof is analogous to the one given in the average complexity section.

Average complexity

Even if pivots aren't chosen randomly, quicksort still requires only $O(n \log n)$ time averaged over all possible permutations of its input. Because this average is simply the sum of the times over all permutations of the input divided by n factorial, it's equivalent to choosing a random permutation of the input. When we do this, the pivot choices are essentially random, leading to an algorithm with the same running time as randomized quicksort.

More precisely, the average number of comparisons over all permutations of the input sequence can be estimated accurately by solving the recurrence relation:

$$C(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(C(i) + C(n-i-1)) = 2n\ln n = 1.39n\log_2 n.$$

Here, $n-1$ is the number of comparisons the partition uses. Since the pivot is equally likely to fall anywhere in the sorted list order, the sum is averaging over all possible splits.

This means that, on average, quicksort performs only about 39% worse than the ideal number of comparisons, which is its best case. In this sense it is closer to the best case than the worst case. This fast average runtime is another reason for quicksort's practical dominance over other sorting algorithms.

Space complexity

The space used by quicksort depends on the version used.

Quicksort has a space complexity of $O(\log n)$, even in the worst case, when it is carefully implemented ensuring the following two properties:

- in-place partitioning is used. This requires $O(1)$ space.

- After partitioning, the partition with the fewest elements is (recursively) sorted first, requiring at most $O(\log n)$ space. Then the other partition is sorted using tail recursion or iteration, which doesn't add to the call stack. This idea, as discussed above, was described by R. Sedgewick.[4][5]

The version of quicksort with in-place partitioning uses only constant additional space before making any recursive call. However, if it has made $O(\log n)$ nested recursive calls, it needs to store a constant amount of information from each of them. Since the best case makes at most $O(\log n)$ nested recursive calls, it uses $O(\log n)$ space. The worst case makes $O(n)$ nested recursive calls, and so needs $O(n)$ space; Sedgewick's improved version using tail recursion requires $O(\log n)$ space in the worst case.

We are eliding a small detail here, however. If we consider sorting arbitrarily large lists, we have to keep in mind that our variables like *left* and *right* can no longer be considered to occupy constant space; it takes $O(\log n)$ bits to index into a list of n items. Because we have variables like this in every stack frame, in reality quicksort requires $O((\log n)^2)$ bits of space in the best and average case and $O(n \log n)$ space in the worst case. This isn't too terrible, though, since if the list contains mostly distinct elements, the list itself will also occupy $O(n \log n)$ bits of space.

The not-in-place version of quicksort uses $O(n)$ space before it even makes any recursive calls. In the best case its space is still limited to $O(n)$, because each level of the recursion uses half as much space as the last, and

$$\sum_{i=0}^{\infty} \frac{n}{2^i} = 2n.$$

Its worst case is dismal, requiring

$$\sum_{i=0}^{n} (n-i+1) = O(n^2)$$

space, far more than the list itself. If the list elements are not themselves constant size, the problem grows even larger; for example, if most of the list elements are distinct, each would require about $O(\log n)$ bits, leading to a best-case $O(n \log n)$ and worst-case $O(n^2 \log n)$ space requirement.

Selection-based pivoting

A selection algorithm chooses the kth smallest of a list of numbers; this is an easier problem in general than sorting. One simple but effective selection algorithm works nearly in the same manner as quicksort, except that instead of making recursive calls on both sublists, it only makes a single tail-recursive call on the sublist which contains the desired element. This small change lowers the average complexity to linear or $O(n)$ time, and makes it an in-place algorithm. A variation on this algorithm brings the worst-case time down to $O(n)$ (see selection algorithm for more information).

Conversely, once we know a worst-case $O(n)$ selection algorithm is available, we can use it to find the ideal pivot (the median) at every step of quicksort, producing a variant with worst-case $O(n \log n)$ running time. In practical implementations, however, this variant is considerably slower on average.

Another variant is to choose the Median of Medians as the pivot element instead of the median itself for partitioning the elements. While maintaining the asymptotically optimal run time complexity of $O(n \log n)$ (by preventing worst case partitions), it is also considerably faster than the variant that chooses the median as pivot.

Variants

There are three well known variants of quicksort:

- **Balanced quicksort:** choose a pivot likely to represent the middle of the values to be sorted, and then follow the regular quicksort algorithm.
- **External quicksort:** The same as regular quicksort except the pivot is replaced by a buffer. First, read the M/2 first and last elements into the buffer and sort them. Read the next element from the beginning or end to balance writing. If the next element is less than the least of the buffer, write it to available space at the beginning. If greater than the greatest, write it to the end. Otherwise write the greatest or least of the buffer, and put the next element in the buffer. Keep the maximum lower and minimum upper keys written to avoid resorting middle elements that are in order. When done, write the buffer. Recursively sort the smaller partition, and loop to sort the remaining partition.
- **Three-way radix quicksort** (also called **multikey quicksort**): is a combination of radix sort and quicksort. Pick an element

from the array (the pivot) and consider the first character (key) of the string (multikey). Partition the remaining elements into three sets: those whose corresponding character is less than, equal to, and greater than the pivot's character. Recursively sort the "less than" and "greater than" partitions on the same character. Recursively sort the "equal to" partition by the next character (key).

Comparison with other sorting algorithms

Quicksort is a space-optimized version of the binary tree sort. Instead of inserting items sequentially into an explicit tree, quicksort organizes them concurrently into a tree that is implied by the recursive calls. The algorithms make exactly the same comparisons, but in a different order.

The most direct competitor of quicksort is heapsort. Heapsort's worst-case running time is always $O(n \log n)$. But, heapsort is assumed to be on average somewhat slower than quicksort. This is still debated and in research, with some publications indicating the opposite.[12][13] In Quicksort remains the chance of worst case performance except in the introsort variant, which switches to heapsort when a bad case is detected. If it is known in advance that heapsort is going to be necessary, using it directly will be faster than waiting for introsort to switch to it.

Quicksort also competes with mergesort, another recursive sort algorithm but with the benefit of worst-case $O(n \log n)$ running time. Mergesort is a stable sort, unlike quicksort and heapsort, and can be easily adapted to operate on linked lists and very large lists stored on slow-to-access media such as disk storage or network attached storage. Although quicksort can be written to operate on linked lists, it will often suffer from poor pivot choices without random access. The main disadvantage of mergesort is that, when operating on arrays, it requires $O(n)$ auxiliary space in the best case, whereas the variant of quicksort with in-place partitioning and tail recursion uses only $O(\log n)$ space. (Note that when operating on linked lists, mergesort only requires a small, constant amount of auxiliary storage.)

Bucket sort with two buckets is very similar to quicksort; the pivot in this case is effectively the value in the middle of the value range, which does well on average for uniformly distributed inputs.

See also

- qsort
- Introsort
- Flashsort

Notes

1. ^ S. S. Skiena, The Algorithm Design Manual, Second Edition, Springer, 2008, p. 129
2. ^ "Data structures and algorithm: Quicksort" (http://www.cs.auckland.ac.nz/~jmor159/PLDS210/qsort1a.html) . Auckland University. http://www.cs.auckland.ac.nz/~jmor159/PLDS210/qsort1a.html.
3. ^ "An Interview with C.A.R. Hoare" (http://cacm.acm.org/magazines/2009/3/21782-an-interview-with-car-hoare/fulltext) . Communications of the ACM, March 2009 ("premium content"). http://cacm.acm.org/magazines/2009/3/21782-an-interview-with-car-hoare/fulltext.
4. ^ *a b* R. Sedgewick, Algorithms in C, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching, 3rd Edition, Addison-Wesley
5. ^ *a b* R. Sedgewick, Implementing quicksort programs, Comm. ACM, 21(10):847-857, 1978.
6. ^ qsort.c in GNU libc: [1] (http://www.cs.columbia.edu/~hgs/teaching/isp/hw/qsort.c) , [2] (http://www.google.com/codesearch/p?hl=en#p9nGS4eQGUI/pub/gnu/glibc/glibc-2.5.tar.bz2%7CHT3Jwvgod1I/glibc-2.5/stdlib/qsort.c&q=package:glibc%20qsort%20package:%22ftp://ftp.gnu.org/pub/gnu/glibc/glibc-2.5.tar.bz2%22)
7. ^ http://home.tiscalinet.ch/t_wolf/tw/ada95/sorting/index.html
8. ^ http://www.ugrad.cs.ubc.ca/~cs260/chnotes/ch6/Ch6CovCompiled.html
9. ^ R.Miller, L.Boxer, Algorithms Sequential & Parallel, A Unified Approach, Prentice Hall, NJ, 2006
10. ^ David M. W. Powers, Parallelized Quicksort and Radixsort with Optimal Speedup (http://citeseer.ist.psu.edu/327487.html) , *Proceedings of International Conference on Parallel Computing Technologies*. Novosibirsk. 1991.
11. ^ M. D. McIlroy. A Killer Adversary for Quicksort. Software Practice and Experience: vol.29, no.4, 341–344. 1999. At Citeseer (http://citeseer.ist.psu.edu/212772.html)
12. ^ Hsieh, Paul (2004). "Sorting revisited." (http://www.azillionmonkeys.com/qed/sort.html) . www.azillionmonkeys.com. http://www.azillionmonkeys.com/qed/sort.html. Retrieved 2010-04-26.
13. ^ MacKay, David (2005-12-01). "Heapsort, Quicksort, and Entropy" (http://users.aims.ac.za/~mackay/sorting/sorting.html) . users.aims.ac.za/~mackay. http://users.aims.ac.za/~mackay/sorting/sorting.html. Retrieved 2010-04-26.

References

- R. Sedgewick, Implementing quicksort programs, Comm. ACM, 21(10):847-857, 1978. Implementing Quicksort Programs (http://delivery.acm.org/10.1145/360000/359631/p847-sedgewick.pdf?key1=359631&key2=9191985921&coll=DL&dl=ACM&CFID=6618157&CFTOKEN=73435998)
- Brian C. Dean, "A Simple Expected Running Time Analysis for Randomized 'Divide and Conquer' Algorithms." *Discrete Applied Mathematics* 154(1): 1-5. 2006.
- Hoare, C. A. R. "Partition: Algorithm 63," "Quicksort: Algorithm 64," and "Find: Algorithm 65." Comm. ACM 4(7), 321-322, 1961
- Hoare, C. A. R. "Quicksort." (http://dx.doi.org/10.1093/comjnl/5.1.10) Computer Journal 5 (1): 10-15. (1962). (Reprinted in Hoare and Jones: *Essays in computing science* (http://portal.acm.org/citation.cfm?id=SERIES11430.63445) , 1989.)
- David Musser. Introspective Sorting and Selection Algorithms, Software Practice and Experience vol 27, number 8, pages 983-993, 1997
- Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Pages 113–122 of section 5.2.2: Sorting by Exchanging.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 7: Quicksort, pp. 145–164.
- A. LaMarca and R. E. Ladner. "The Influence of Caches on the Performance of Sorting." Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 1997. pp. 370–379.
- Faron Moller. Analysis of Quicksort (http://www.cs.swan.ac.uk/~csfm/Courses/CS_332/quicksort.pdf) . CS 332: Designing Algorithms. Department of Computer Science, Swansea University.
- Conrado Martínez and Salvador Roura, *Optimal sampling strategies in quicksort and quickselect.* SIAM J. Computing 31(3):683-705, 2001.
- Jon L. Bentley and M. Douglas McIlroy, Engineering a Sort Function (http://citeseer.ist.psu.edu/bentley93engineering.html) , *Software—Practice and Experience*, Vol. 23(11), 1249–1265, 1993

External links

- Animated Sorting Algorithms: Quick Sort (http://www.sorting-algorithms.com/quick-sort) – graphical demonstration and discussion of quick sort
- Animated Sorting Algorithms: 3-Way Partition Quick Sort (http://www.sorting-algorithms.com/quick-sort-3-way) – graphical demonstration and discussion of 3-way partition quick sort
- Interactive Tutorial for Quicksort (http://pages.stern.nyu.edu/~panos/java/Quicksort/index.html)
- Javascript Quicksort and Bubblesort (http://thomasgilray.com/classes/quicksort.php)
- Quicksort applet (http://www.atkinson.yorku.ca/~sychen/research/sorting/sortingHome.html) with "level-order" recursive calls to help improve algorithm analysis
- Multidimensional quicksort in Java (http://fiehnlab.ucdavis.edu/staff/wohlgemuth/java/quicksort)
- Literate implementations of Quicksort in various languages (http://en.literateprograms.org/Category:Quicksort) on LiteratePrograms
- Quicksort tutorial with illustrated examples (http://www.mycstutorials.com/articles/sorting/quicksort)
- A colored graphical Java applet (http://coderaptors.com/?QuickSort) which allows experimentation with initial state and shows statistics

Retrieved from "http://en.wikipedia.org/wiki/Quicksort"
Categories: Sorting algorithms | Comparison sorts | Articles with example pseudocode | 1961 in science

- This page was last modified on 15 September 2011 at 14:20.
- Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. See Terms of use for details.
  Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

6.005 Lecture 4

9/19

State Machines

- [ ] mutable objects
- [ ] aliasing
- [ ] state machines

PS2 out, due Thursday night @ midnight

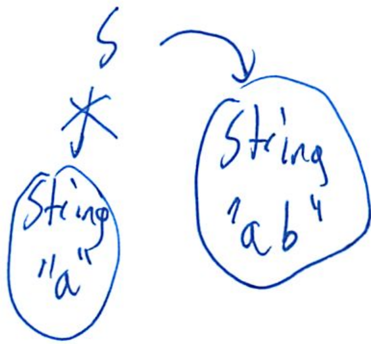
Do the EECS UG survey ~MIDI

So far just wrote procedures
- nothing changes from call to call
- called functional programming
- can reason about things locally

String s = "a";

s += "b"; // s = s + "b";
//s = s.concat("b");

(2) Constructs a new string



Since string is immutable
- easier to reason about
- draw double circle

Is a mutable string, called StringBuilder

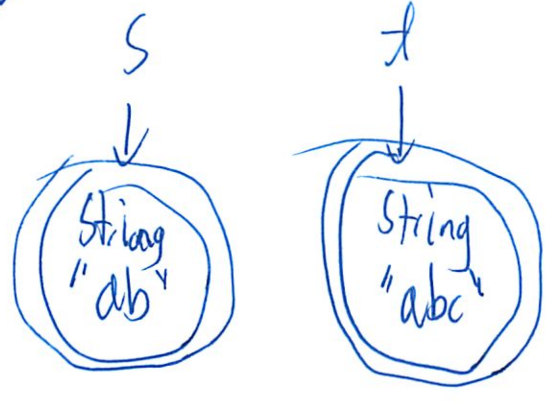
```
sb = new SB("a");
sb.append("b")
```

It really does not matter when you only have 1 reference
But when have > 1

③

String t = s;
String t += "c";

SB tb = sb;
sb.append("c")



Benefit/Risk of mutable data objects

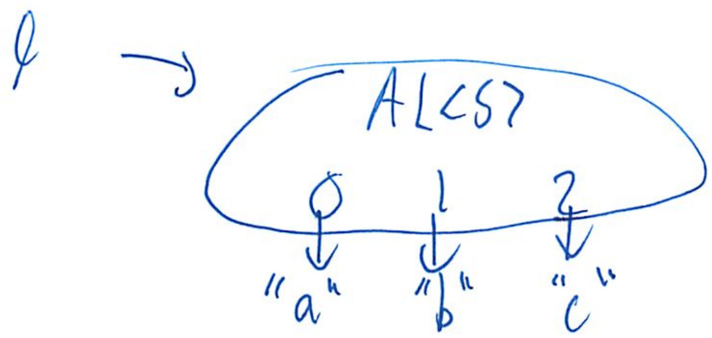
Lots of overhead to make new objects
- like if making a loop building a string
- always a new object $O(n^2)$

Faster in string builder - linear $O(n)$

Iterators

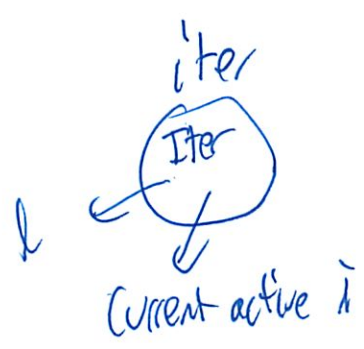
```
List<String> l = ...
for (String s : l) {
  ;
  ;
}
```

④



The for is a Iterator
- a mutable object

```
Iterator<String> iter = l.iterator()
```




```
while (iter.hasNext()) {
    String s = iter.next();
    ...
}
```

Use this syntax for more control/power like when you need current active i

(5)

To write new method class

1. Write Spec
2. Write Test
3. Implement

final - this ~~object~~ always points to same obj in memory

that item can still change

Symbol ⟶

private - only methods in ~~that~~ file can read/write

Constructor - runs when call new

- no return type!
- takes parameter
- named same as class

this is always passed implicitly

- don't need to ~~say~~ include
- also don't need to say this.i

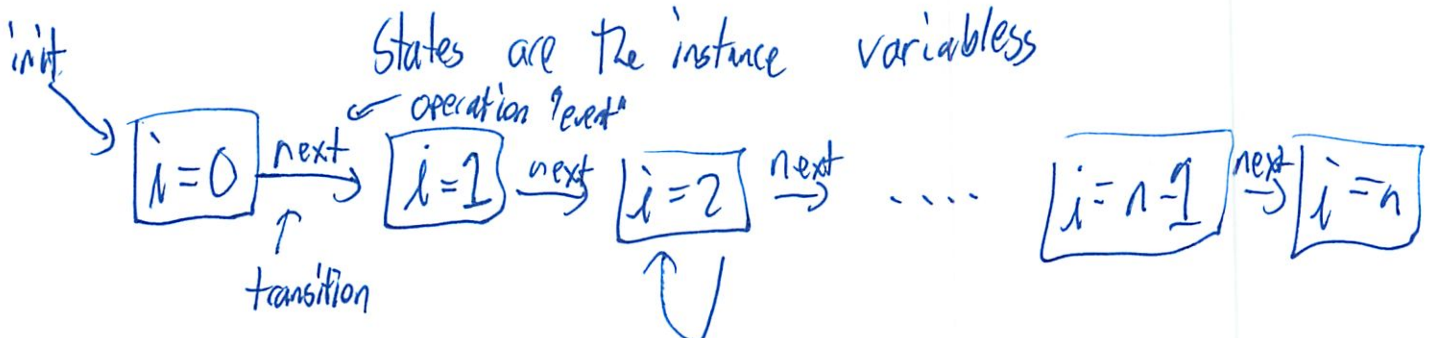
⑥

As long as unambiguous

Static - ~~an~~ does not matter on instance
Can't use this
Since no concept of that

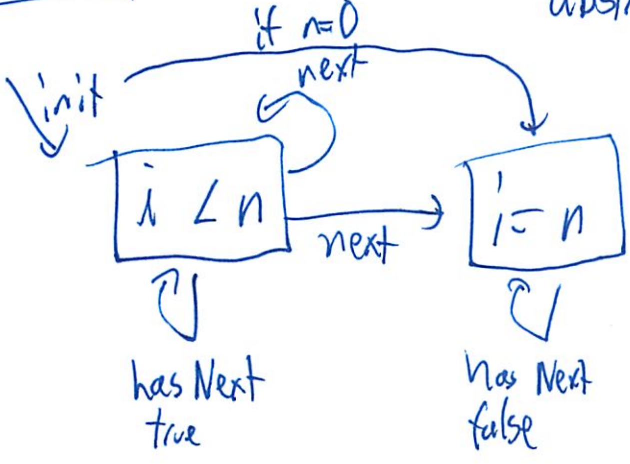
Can draw State Machine (SM) of Iter

States are the instance variabless



hasNext
? in every state
but does not
cause any state
changes - so often
abstract away

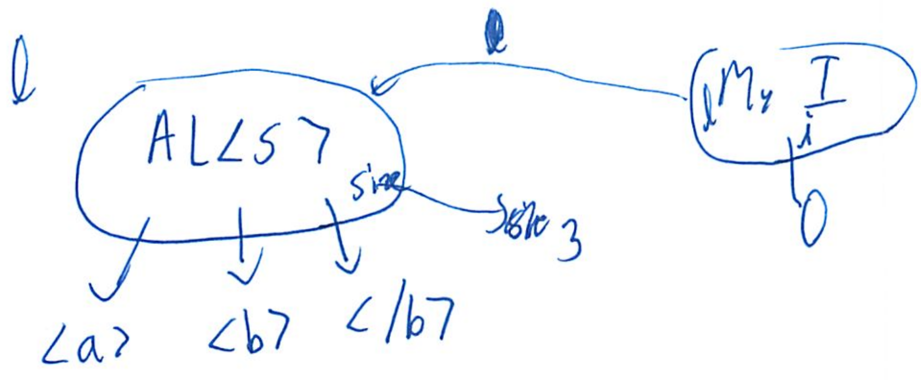
Abstract away



this model/abstraction
is non deterministic
but actual one is
deterministic

So build HTML parser

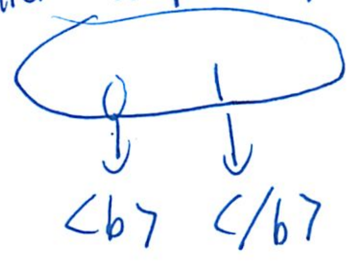
But its not what we expected - why?



Then removes element 0

Now all the elements change

Iterator bumps up



Now we've skipped something

Can decrement i when removing something

But we can have multiple iterators - tricky to adjust all

The built in iterator in for() has same problem

└ fail fast exception: ConcurrentMod Exception

(8)

So global requirement: don't modify collection while iterating!

So when testing need to worry about <u>sequences</u> of operations
- touch every event in SM
- all-actions coverage (next, hasNext)

A better thing is to touch every state
- next once

But does not cover all transitions

An even better: all transitions
- draw state diagram to see them

n=2 hasNext i=0
next i=1
next i=2
hasNext i=2

Stronger: All possible paths
- usually infeasible
- sometimes infinite

6.005 Software Construction
Fall 2011
Prof. Rob Miller

L4: State Machines

Today

- Mutable objects
- Aliasing
- State machines

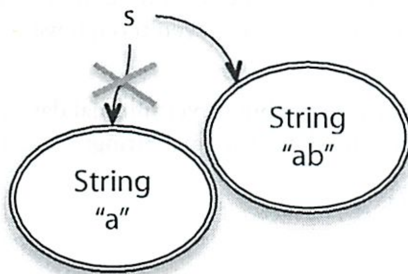
Required reading (from the Java Tutorial)

- Classes and Objects, including:
    - Classes
    - Objects
    - More on Classes
    - Nested Classes
    - Enum Types
- the static keyword
- the final keyword

Mutable objects

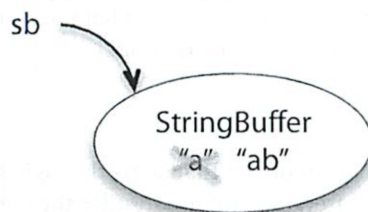
String is immutable: once created, a String object always has the same value. To add something to the end of a String, you have to create a new String object

```
String s = "a";
s = s.concat("b");    /// s = s + "b" is syntactic sugar for this
```



StringBuilder (another builtin Java class) is a **mutable** object that represents a string of characters. It has operations that change the value of the object, rather than just returning new values:

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def");
```

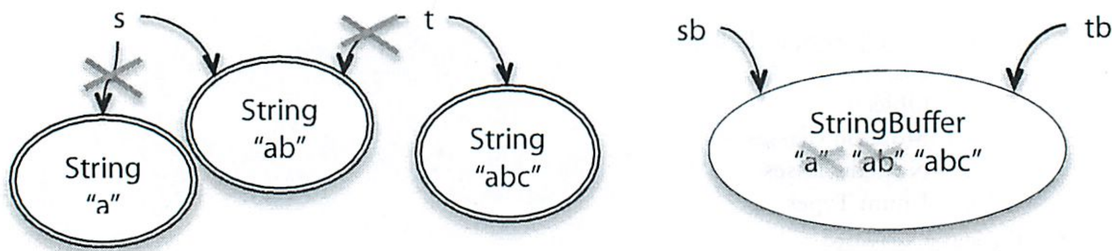


These lecture notes have been collaboratively authored, with contributions from Saman Amarasinghe, Srini Devadas, Michael Ernst, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. *Copyright © MIT.*

StringBuilder has other **mutator** operations as well, for deleting parts of the string, inserting in the middle, or changing individual characters.

So what? In both cases, you end up with s and sb referring to the string of characters abcdef. The difference between mutability and immutability doesn't matter much when there's only one reference to the object. But there are big differences in how they behave when there are *other* references to the object, like t and tb introduced below:

```
String t = s;
t = t + "c";

StringBuilder tb = sb;
tb.append("c");
```



Why do we need the mutable StringBuilder in programming? A common use for it is to concatenate a large number of strings together, like this:

```
String s = "";
for (int i = 0; i < n; ++i) {
        s = s + n;
}
```

Using immutable Strings, this makes a lot of temporary copies – the first number of the string ("0") is actually copied *n* times in the course of building up the final string, the second number is copied *n-1* times, and so on. It actually costs $O(n^2)$ time just to do all that copying, even though we only concatenated *n* elements.

StringBuilder is designed to minimize this copying. It uses a simple but clever internal data structure to avoid doing any copying at all until the very end, when you ask for the final String:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < n; ++i) {
        sb.append(String.valueOf(n));
}
String s = sb.toString();
```

Getting good performance is one reason why we use mutable objects. Another is convenient sharing: two parts of your program can communicate more conveniently by sharing a common mutable data structure.

But using mutable data comes with big risks – it becomes harder to understand what your program is doing, and much harder to enforce contracts. We'll look at an example of that next.

Iterating over collections

The next mutable object we're going to look at is an iterator – an object that steps through a collection of elements and returns the elements one by one. Iterators are used under the covers in Java when you're using a for loop to step through a List or array. This code:

```
List<String> l = ...;
for (String s : l) {
      System.out.println(s);
}
```

actually unpacks into something like this:

```
List<String> l = ...;
Iterator iter = l.iterator();
while (l.hasNext()) {
      String s = iter.next();
      System.out.println(s);
}
```

An iterator has two methods: next() returns the next element in the collection; and hasNext() tests whether the iterator has reached the end of the collection. Note that the next() method is a **mutator**, not only returning an element but also advancing the iterator so that the subsequent call to next() will return a different element.

To better understand how an iterator works, here's a simple implementation of an iterator for ArrayList<String>:

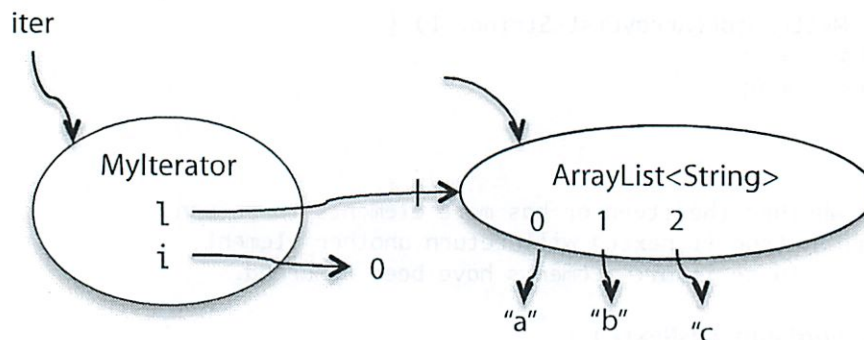
```
/**
 * A MyIterator is a mutable object that iterates over
 * the elements of an ArrayList<String>, from first to last.
 * This is just an example to show how an iterator works.
 * In practice, you should use the ArrayList's own iterator object,
 * returned by its iterator() method.
 */
public class MyIterator {

    private final ArrayList<String> l;
    private int i;
    // l[i] is the next element that will be returned by next();
    // i == l.size() means no more elements to return

    /**
     * Make an iterator.
     * @param l list to iterate over
     */
    public MyIterator(ArrayList<String> l) {
        this.l = l;
        this.i = 0;
    }

    /**
     * Test whether the iterator has more elements to return.
     * @return true if next() will return another element,
     *         false if all elements have been returned.
     */
    public boolean hasNext() {
        return i < l.size();
    }
```

```
    /**
     * Get the next element of the list.
     * Requires: hasNext() returns true.
     * Modifies: this iterator to advance it to the element
     *           following the returned element.
     * @return next element of the list
     */
    public String next() {
        final String s = l.get(i);
        ++i;
        return s;
    }
}
```

MyIterator makes use of a few Java language features that are different from the classes we've been writing to this point. Make sure you read the Java Tutorial sections required for this lecture so that you understand them:

- *instance variables*, also called fields in Java. Instance variables differ from method parameters and local variables; the instance variables are stored in the object instance and persist for longer than a method call. What are the instance variables of MyIterator?
- a *constructor*, which makes a new object instance and initializes its instance variable. Where is the constructor of MyIterator?
- the *static* keyword is missing from MyIterator's methods, which means they are instance methods that must be called on an instance of the object, e.g. iter.next().
- the *this* keyword is used at one point to refer to the instance object, in particular to refer to an instance variable (*this.l*). This was done to disambiguate two different variables named *l* (an instance variable and a constructor parameter). Most of MyIterator's code refers to instance variables without an explicit *this*, but this is just a convenient shorthand that Java supports -- e.g., *i* actually means *this.i*.
- *private* is used for the object's internal state and internal helper methods, while *public* indicates methods and constructors that are intended for clients of the class.
- *final* is used to indicate which parts of the object's internal state can change and which can't. *i* is allowed to change (next() updates it as it steps through the list), but *l* cannot (the iterator has to keep pointing at the same list for its entire life – if you want to iterate through another list, you're expected to create another iterator object).

Here's a snapshot diagram showing a typical state for a MyIterator object in action:



Note that we drew a slash across the arrow from *l*, to indicate that it's *final*. That means that the arrow can't change once it's drawn. But the ArrayList object it points to is mutable – elements can be changed within it -- and declaring *l* as final has no effect on that.

Why do iterators exist? There are many kinds of collection data structures (linked lists, maps, hash tables) with different kinds of internal representations. The iterator concept allows a single uniform way to access them all, so that client code is simpler and the collection implementation can change without changing the client code. Most modern languages (including Python, C#, and Ruby) use the notion of an iterator. It's an effective **design pattern** (a well-tested solution to a common design problem). We'll see many other design patterns as we move through the course.

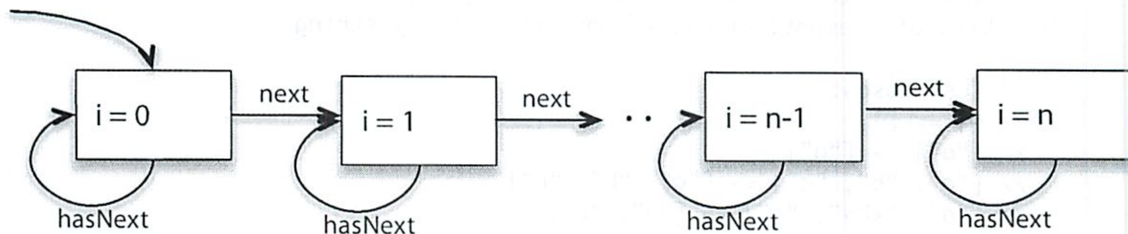
Regarding a mutable object as a state machine

One useful perspective for regarding mutable is as a *state machine.* A state machine is a set of *states* that a system can be in (drawn as boxes) with the possible transitions between them (drawn as edges). The transitions are called *events.*

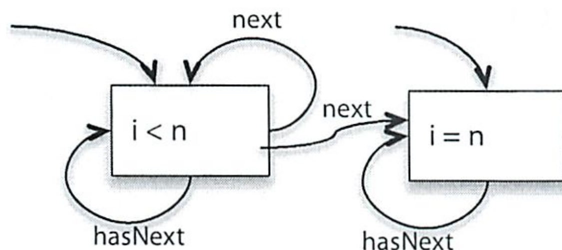
For a mutable object, the state is represented by the **instance variables** of the object – particularly the ones that can be mutated over the course of the object's lifetime. The state of a MyIterator object is represented by *i*, the index of its current position in the list.

The events are the public operations that can be performed on the object -- the methods that can be called. These represent events arriving from the outside world (from a client using the object) that cause transitions in the object's internal state. For a MyIterator, the methods are hasNext and next. Of these two methods, next is a **mutator**, however, so it can cause transitions to a new state. hasNext is an **observer** – it returns information about the current state, but never causes a transition to a new state.

Here's a very explicit state machine diagram for a MyIterator object over a list of length *n* (*n > 0*), showing all the possible states that the object can experience:



In practice we will find it more useful to draw more abstract state diagrams that combine multiple states together:



Here we've combined all the states where i<n into a single state, and combined their transitions as well. Note that we now have a *nondeterministic* state machine – the *next* event (method call) can transition to either i<n (if we still have more elements to return) or to i=n (if we've run to the end of the list). In this case, the nondeterminism was introduced by our decision to abstract away from the particular value of i. The behavior of the MyIterator object is still completely deterministic; it knows which particular value of i it has. We've just omitted it from this picture. Other state machines might be genuinely nondeterministic, in the sense that the transition chosen in response to an event might be random.

This state machine also correctly handles the case where *n* (the length of the list is zero), by having two initial-state transitions. Either $i<n$ or $i=n$ may be the starting state of the machine.

The risk of mutation

Let's try using our iterator for a simple job. Suppose we have a list of tokens extracted from a web page, so the tokens include both words ("hello" and "world") and HTML tags ("<a>", "<br>", etc.). We want a method *stripTags* that will delete the HTML tags from the list, leaving the words behind. Following good practices, we first write the spec:

```
/**
 * Remove html tags from a list.
 * Modifies l by removing elements of the form "<*>".
 * @param l list of words and html tags.
 */
public static void stripTags(ArrayList<String> l) {...}
```

Note that stripTags has a frame condition (*modifies* clause) in its contract, warning the client that its list argument will be mutated.

Next, following test-first programming, we devise a testing strategy that partitions the input space, and choose test cases to cover that partition:

```
// Testing strategy:
//   l.size: 0, 1, n
//   contents: no tags, one tag, all tags
//   position: tag at start, tag in middle, tag at end
//   kind of element: <foo>, </foo>, word, empty string

// Test cases:
// [] => []
// ["a"] => ["a"]
// ["a", "b", "c"] => ["a", "b", "c"]
// ["a", "<b>", "c"] => ["a", "c"]
// ["<a>", "<b>", "<c>"] => []
```

Finally we implement it:

```
public static void stripTags(ArrayList<String> l) {
    MyIterator iter = new MyIterator(l);
    while (iter.hasNext()) {
        String s = iter.next();
        if (isTag(s)) {
            l.remove(s);
        }
    }
}

// returns true iff s is an html tag of the form "<*>" for any *
private static boolean isTag(String s) {
    return s.startsWith("<") && s.endsWith(">");
}
```

(Note that we pulled out the test for whether a token is HTML into a separate method, isTag. This improves the readability of stripTags, and allows us to make isTag more complicated if necessary, since HTML can be very complicated, and test it independently of stripTags.)

Now we run our test cases, and they work! ... almost. The last test case fails:

```
// stripTags(["<a>", "<b>", "<c>"])
//    expected [], actual ["<b>"]
```

We got the wrong answer: stripTags left a tag behind in the list. Why? Trace through what happens. It will help to use a snapshot diagram showing the MyIterator object and the ArrayList object and update it while you work through the code.

Note that this isn't just a bug in our MyIterator. The built-in iterator in ArrayList suffers from the same problem, and so does the for() loop that's syntactic sugar for it. The problem just has a different symptom. If you used this code instead:

```
for (String s : l) {
    if (isTag(s)) {
        l.remove(s);
    }
}
```

then you'll get a ConcurrentModificationException. The builtin iterator detects that you're changing the list under its feet, and cries foul. (How do you think it does that?)

How can you fix this problem? One way is to use the remove() method of Iterator, so that the iterator adjusts its index appropriately:

```
Iterator iter = l.iterator();
while (iter.hasNext()) {
    String s = iter.next();
    if (isTag(s)) {
        iter.remove(s);
    }
}
```

(This is actually more efficient as well, it turns out, because iter.remove() already knows where the element it should remove is, while l.remove() had to search for it again.)

But this doesn't fix the whole problem. What if there are *other* Iterators currently active over the same list? They won't all be informed!

Aliasing

This is a fundamental issue with mutable data structures. Multiple references to the same mutable object (also called *aliases* for the object) may mean that multiple places in your program – possibly widely separated – are relying on that object to remain consistent.

To put it in terms of specifications, contracts can't be enforced in just one place anymore, e.g. between the client of a class and the implementer of a class. Contracts involving mutable objects now depend on the good behavior of everyone who has a reference to the mutable object. (As a symptom of this non-local contract phenomenon, consider the Java collections classes, which are normally documented with very clear contracts on the client and implementer of a class. Try to find where it documents this particular requirement on the client – that you can't modify a collection while you're iterating over it. Who takes responsibility for it? Iterator? List? Collection?)

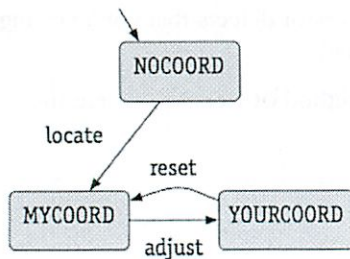
The need to reason about global properties like this make it much harder to understand, and be confident in the correctness of, programs with mutable data structures. We still have to do it – for performance and convenience – but we pay an big cost in bug safety for doing so.

Using state machines for analysis and design

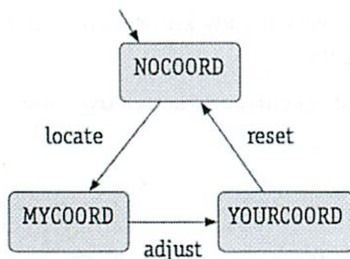
State machine notation gives us a way to think about the behavior of a class, module, or entire system, so that we can understand it, communicate it, and analyze it.

Here's an example. In Afghanistan, December 2001: a US soldier uses a "plugger" (PLGR: precision lightweight global-positioning satellite receiver) to mark a Taliban position for an air-strike. The normal operation of this device starts with a **locate** operation that determins the user's own coordinates (from GPS satellites), and then an **adjust** operation that adjusts by a distance and direction (obtained by sighting from the user's position to the target) to obtain the target's coordinates. In this incident, the soldier noticed a battery-low warning, so he replaced the battery, and then radioed in the coordinates he read from the screen. The resulting airstrike hit his own position, killing him and two comrades and wounding 20 others.

What happened? Replacing the battery reset the device to displaying the *user's* position, forgetting the adjustment to the target. The state machine for the device's display looked like this:



whereas a much safer user interface would have done this instead:



Vernon Loeb. "Friendly Fire Deaths Traced to Dead Battery; Taliban Targeted, but U.S. Forces Killed." *Washington Post.* March 24, 2002

State machines are a simple notation for describing behaviors which is succinct and abstract – but not vague! We can use it as a basis for analysis and description of problems like these, and also for implementation. We'll see later in this lecture how state machines are useful for choosing test cases.

An example: a MIDI piano

State machines are often used for input and output. Java's InputStream and OutputStream objects are simple state machines that read and write streams of characters (such as files or network connections).

Let's look at two other kinds of input and output: your keyboard, and a MIDI music synthesizer.

A single keyboard key has two states

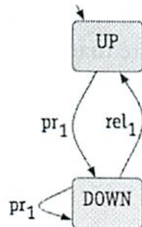
UP: the key is up (initial state)

DOWN: the key is held down

There are two kinds of input events, with these designations:

pr: the keyboard driver reports a key press

rel: the keyboard driver reports a release



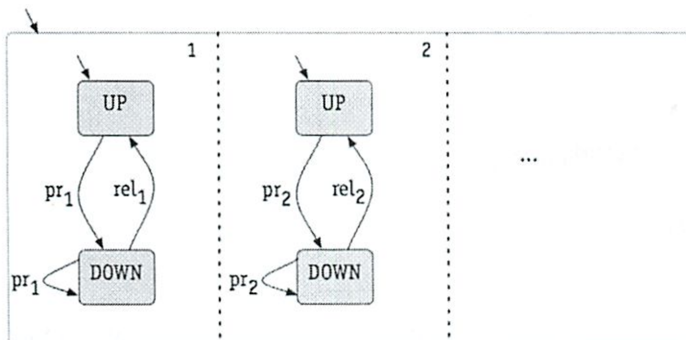
(This keyboard driver reports multiple press events while the key is held down, so we have a pr self-transition in the DOWN state.)

The meaning of a state machine is the **set of traces** that it accepts, where a trace is a sequence of events. The traces of this machine are:

<> (empty trace)
<pr>
<pr, rel>
<pr, pr, rel>

...

An entire keyboard can be represented as a parallel combination of state machines, which we denote graphically as follows:



Each key's machine can take steps independently. As a general rule, **shared** events (transitions with the same label) must be synchronized, but there is no sharing here, because each state machine's events are labeled by the key (1, 2, ...).

The traces of this machine include:

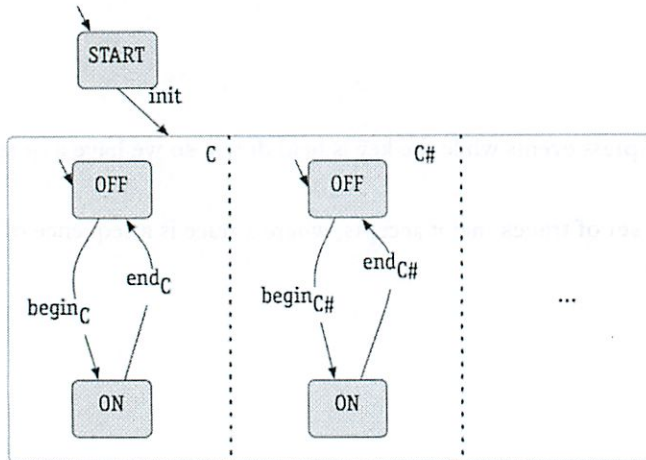
<>
<pr1>
<pr1, rel1>

<pr1, rel1, pr>
<pr1, pr2, pr1>

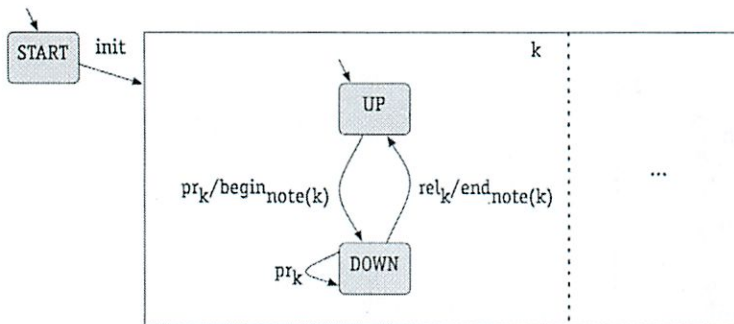
...

This model is actually an *over-approximation* -- it allows more traces than can really happen to a keyboard. One such trace is <pr1, pr2, pr1> -- try it and see. But this is OK; a design that handles cases that can never arise isn't a problem; it's a design that fails to handle cases that's a problem.

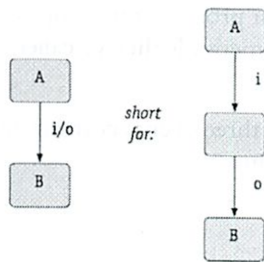
We've looked at the input side of the example. Now let's turn to the output side: the MIDI synthesizer. Unlike a physical keyboard, this is a software system that needs explicit intialization, so we indicate that with an *init* event. Then, we just have a set of parallel state machines, one for each note the synthesizer can play, and events to turn them on or off:



Finally, let's connect them together. We'll draw another state machine that connects the keyboard to the MIDI device:



This machine is using a shorthand on its edges that takes an input event (such as pr_k) and emits an output event ($begin_{note(k)}$). This is just a shorthand for two edges with an intervening state, i.e.:



There's a subtlety here, in that the output *o* need not follow the input *i* immediately. Another event might intervene. This will become a critical issue when we design concurrently-running state machines, which is a topic for a future lecture.

Our final picture shows how presses and releases of the keyboard are related to starting and stopping notes in the MIDI synthesizer, as well as the detail that additional presses in the DOWN state should be ignored. It can be translated more or less directly to code.

State machine semantics

Formally, a state machine consists of:

- a set of states State
  - e.g. State = { UP, DOWN, PRESSED, RELEASED }
- a set of initial states Init
  - e.g. Init = { UP }
- a set of events Event
  - e.g Event = { pr, rel, begin, end }
- a transition relation *trans* ⊆ State × Event × State
  - e.g. *trans* ⊆ State × Event × State
    = { (UP, pr, PRESSED), (PRESSED, begin, DOWN), (DOWN, pr, DOWN), (DOWN, rel, RELEASED), (RELEASED, end, UP) }
- a trace set *traces* (derived from trans and Init)
  - e.g. traces = { <>, <pr>, <pr, begin>, <pr, begin, pr>, <pr, begin, rel>, ...}

Common misconceptions about state machine modeling

A state machine is not a flow chart. There is no behavior or logic inside a state. It doesn't make sense to write a label like "Is the key pressed?" inside a state.

A state machine has no "decision edges," either. Some transitions in the state machine may be nondeterministic (such as the *next* transitions in the Iterator state machine above), and logic in the system may determine which transition to take, but that logic is not expressed in the state machine diagram.

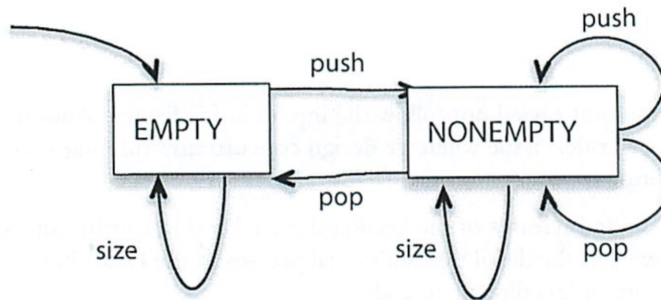
Change doesn't happen within a state. Think of it like a snapshot of the world. Change happens only on the transitions.

Another example: a stack

Let's develop another mutable class and model it as a state machine. A *stack* is a collection of elements that can only be changed at one end: you can *push* an element on top of the stack, and *pop* an element from the top of the stack, but you can't insert or delete elements in the middle or bottom of the stack. Think of a stack like a stack of dinner plates at a buffet. Stacks are useful for keeping

track of work in recursive algorithms (in fact, method calling in most programming languages uses a stack to keep track of the methods that are waiting for results from methods they've called). Stacks are also used in "reverse Polish" calculators, like those made by HP.

Let's draw the state machine for a stack. We'll have two states, and three events corresponding to the *push* and *pop* mutators, plus a *size* method that doesn't change state:



Recall that this is an abstraction for more detailed states: NONEMPTY represents a lot of different stacks with different sizes and different contents.

Note also that we've made a design choice here that pop events are not legal in the empty state.

Again we follow spec-first, test-next, implement-last pattern. Here's our spec:

```
/**
 * A Stack is a mutable object representing a last-in-first-out
 * stack of elements (of an arbitrary type E).
 * Elements can be pushed onto the stack, and then popped off
 * in the reverse order that they were pushed.
 * A Stack can hold an arbitrary number of elements.
 */
public class Stack<E> {

    /**
     * Make a Stack, initially empty.
     */
    public Stack() {...}

    /**
     * Modifies this stack by pushing an element onto it.
     * @param e element to push on top
     */
    public void push(E e) {...}

    /**
     * Modifies this stack by popping off the top element.
     * Requires: stack is not empty, i.e. size() > 0.
     * @return element on top of stack
     */
    public E pop() {...}

    /**
     * @return number of elements in the stack
     */
    public int size() {...}
```

```
}
```

We'll choose an internal state for this stack, and decide how it maps to the Empty and Nonempty states in our state machine:

```
private final List<E> elems = new ArrayList<E>();
// elems contains the elements in the stack,
// in order from oldest pushed (elems[0]) to
//            to the latest item pushed, and the
//             next to be popped (elems[size-1]).
// If elems.size == 0, then the stack is empty.
```

Can you finish the implementation of Stack, by writing the constructor, push, pop, and size methods?

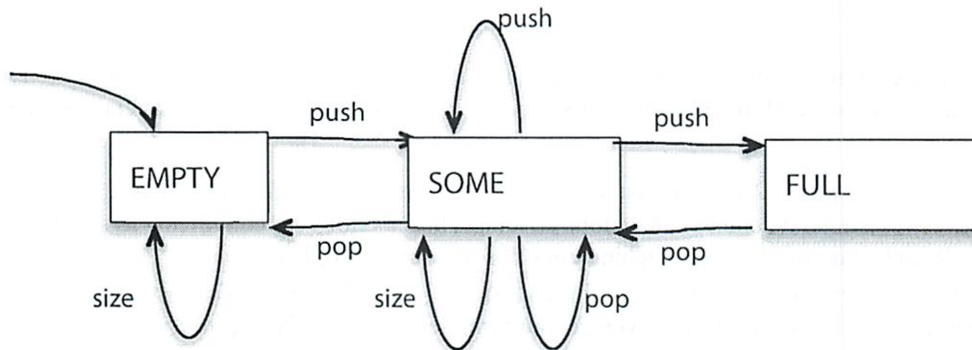
A fixed-size stack

Our stack is flexible, but if we want a truly high-performance stack, it might not be suitable. The ArrayList implementation copies the stack when it gets too big, in order to make room for more elements.

Suppose we know there's a maximum size our stacks will grow, and we're willing to live with that limitation in exchange for fast performance. Then we can use a fixed-size array to implement the stack:

```
private final E[] elems;
private int n;
// elems[] contains the elements in the stack.
// elems[0] is the oldest item pushed;
// elems[n-1] is the latest item pushed, and the
//            next to be popped.
// If n == 0, then the stack is empty.
// If n == elems.length, then the stack is full.
```

The state machine for FixedStack looks like this:



Can you modify the specs for the push, pop, and size methods to match this new design, and implement them?

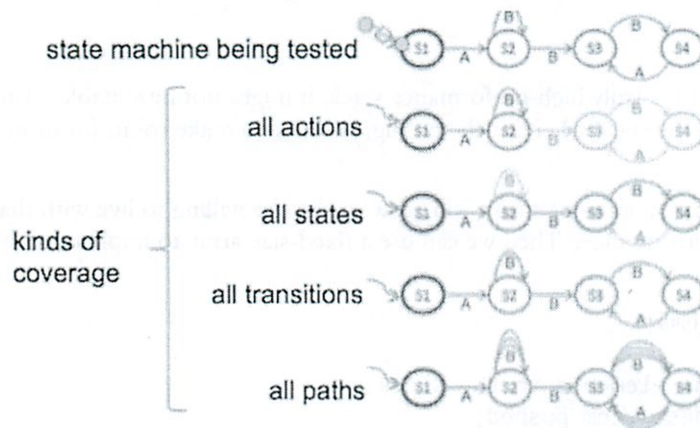
Testing a state machine

For mutable objects, test cases aren't just single inputs – they are *traces*, a sequence of events experienced by the object over its lifetime. For a stack, a single test case would be a sequence of push and pop (and size) method calls.

By drawing the state machine diagram, we can visualize our test cases and see how well they cover the different behaviors of the object. **Coverage** is a measure of test suite quality that refers to the extent to which the tests 'cover' the specification or the implementation.

There are four common kinds of coverage for state machines:

- *all-actions* includes every event in at least some test
- *all-states* visits every state (of the abstract state machine) in at least some test
- *all-transitions* makes every legal transition in the state machine in some test
- *all-paths* explores every possible path of transitions through the state machine.



The four kinds of coverage are related as follows:

- all actions and all-states are weaker than all-transitions
- all-transitions is weaker than all-paths

Consider testing the Iterator state machine below:

i<n   i=n

A test case is a **trace** of events through the state machine, which in this case are method calls like (*hasNext, next, hasNext, next*). Here are some test suites that obtain different levels of coverage of the Iterator state machine:

- all-actions: start with a 2-element list, create a MyIterator, and invoke (*hasNext, next*). This tries every event, so it achieves all-actions coverage, but it's not very convincing as a test suite! Which state of the state machine does this test suite completely miss?
- all states: start with a 1-element list, and then just call (*next*). This touches both states in the diagram, so it achieves all-states coverage, but it misses testing hasNext() entirely.
- all transitions: start with a 2-element list can invoke (*hasNext, next, next, hasNext*). This test crosses every edge in the state machine.

This approach to selecting test cases for a state machine is very similar to the input-space-partitioning technique we've already seen for functions. The state machine partitions the space of object states so that similar states and behaviors are lumped together (this is what we did when we abstracted away

from the detailed Iterator state machine to the more abstract one seen here). Then all-transitions coverage selects test cases that cover the state machine.

Summary

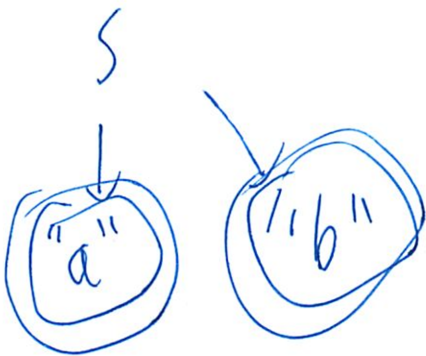
This lecture took a look at mutable objects, and how to represent their behavior as state machines. We saw that mutability is useful for performance and convenience, but it also creates risks of bugs by requiring the code that uses the objects to be well-behaved on a global level, greatly complicating the reasoning and testing we have to do to be confident in its correctness. We also looked at how state machines can be used to represent input and output connections, and how they can help the selection of test cases.

6.005 Recitation 4

9/20

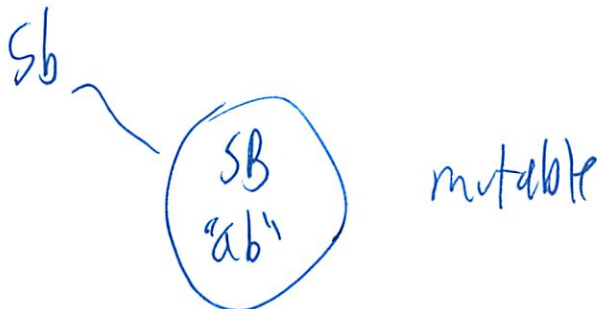
Yesterday Mutability
State Machines

String s = "a";
s = "b";



(I almost know more than the TA here ...)

String Builder



Faster when appending letters
You can prespecify lenght
- extends capacity by 1.5

So if iterating over a list
and delete an element
~~the~~ current i does <u>not</u> adjust! for for
Iterator will error

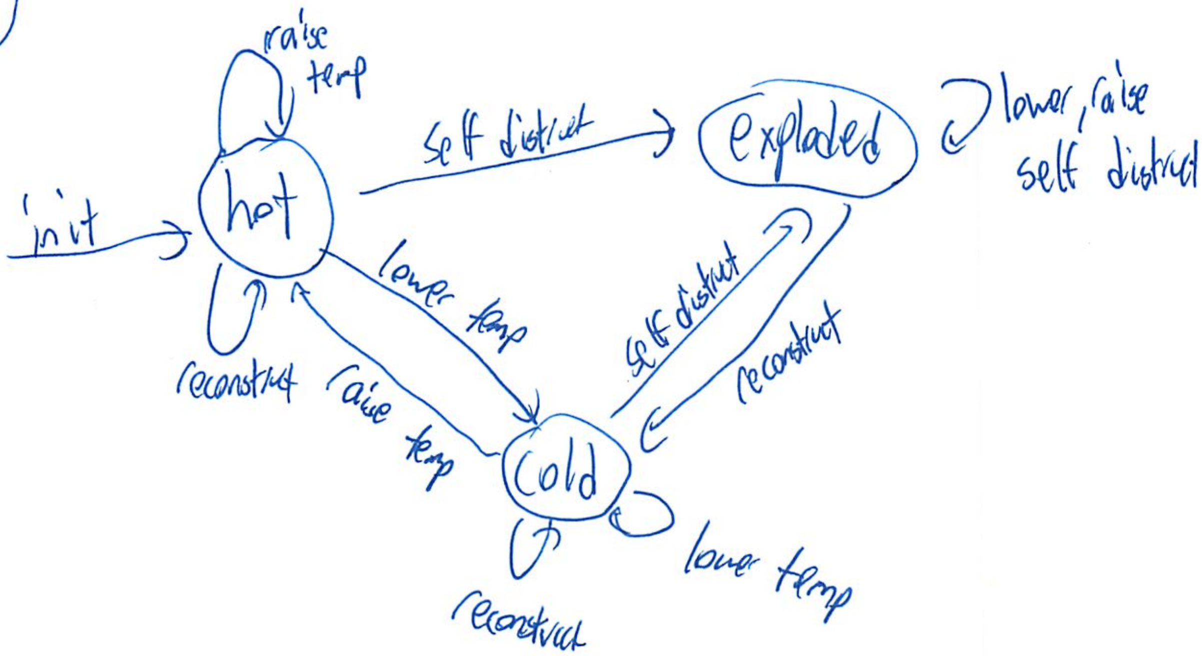
State Machine

Define behavior of your objects

- state
- transitions
- start state
- inputs (conditions for transition)
- outputs

From every state, ~~#~~ specify transitions for every possible input

(3)



What are outputs?

Define a seperate output for each transition

Sometimes output is just what state (like here)

But output can depend on transition

P-Set

Was modified from prior years

So not most beautiful

Have to copy + paste code

Involves threads + locking

About on Piazza

Nick wrote P-Set

(4)

Have to end to end test

hard to unit test

Does it output proper values?

Note is either playing or not playing

2 inputs: key press, key release

6.005 Recitation

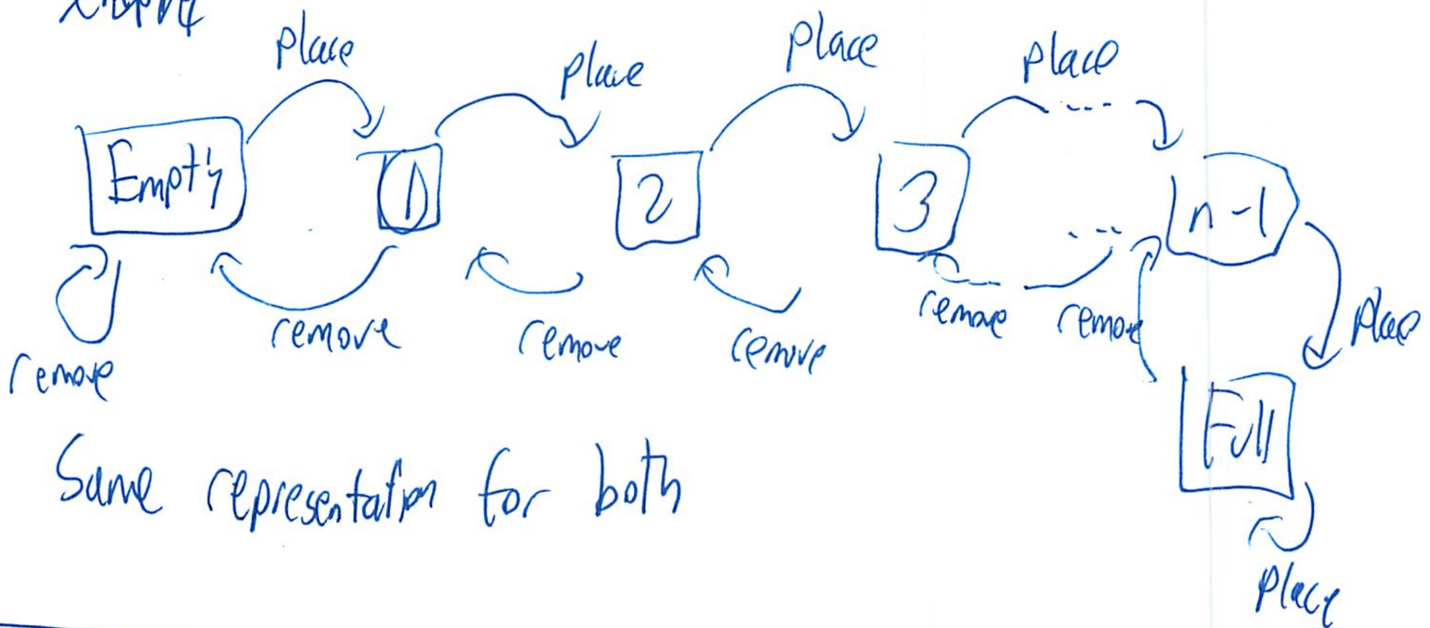
9/22

Stack vs Queue

└ LIFO └ FIFO

Queue + Stack

~~Empty~~



Same representation for both

Office Hours

6.005 Elements of Software Construction | Fall 2011
Problem Set 2: Midi Piano
Due: Thursday, September 22 2011, 11:59 PM

The purpose of this problem set is to give you a chance to design with state machines, and to see one way of translating such a design into code.

The code in the later questions depends on the code in question 1 to work. Questions 2, 3, and 4 can be completed in any order. Question 5 depends on Question 4. Question 4 may take more time to complete than the previous questions.

Do not change the signatures or specifications of any methods, classes, or packages that we have provided you. Your code will be tested automatically, and will break our testing suite if you do so.

To get started, pull out the problem set code from SVN admin.

Overview

Throughout this problem set we will be implementing a simple midi keyboard. We will give it the capability to:

1. Play notes using a row of keys on your computer keyboard
2. Change among a number of instruments
3. Change octaves
4. Record sequences of notes and play them back.

We have provided you with code that wraps a midi device, some abstractions for dealing with musical notes, and some Java Applet code that listens for keypresses. Our applet code calls several methods of PianoMachine; you should keep those methods as the entry points for PianoMachine and not change their signatures or specs (as we will use these functions for automated testing.) Do not modify any code inside the 'midi' or 'piano' packages. For the first four questions, do not modify any code in the PianoApplet class. Please note that we have provided a method, Midi.history(), which gives a text output from the piano and can be used for testing.

Problem 1: Piano Keys

Our midi piano should allow us to play a set of pitches {C, C#, D, ... A#, B} using the keys {'1', '2', ... '-', '='} respectively. When one of these keys is pressed, a note should begin if it isn't already sounding; likewise, when such a key is released, a note should end if it is currently sounding. We provide method signatures for PianoMachine.beginNote and PianoMachine.endNote, which will be triggered by appropriate key presses and releases. Use those methods as entry points in your implementation.

a. [5 points] Write the specs for PianoMachine.beginNote and PianoMachine.endNote, and for any other helper methods you expect to need.

b. [5 points] Write test cases for these methods. You will find Midi.history() useful for this and other tests in this assignment.

c. [15 points] Add this functionality by implementing the beginNote and endNote methods, as well as any helper methods, and adding any necessary state to PianoMachine. To start you off, we've given a provisional implementation of beginNote and endNote which always plays middle 'C'.

Problem 2: Switching Instruments

The midi piano should be able to switch instruments. The 'I' key should switch our instrument mode to the next instrument in a list, or back to the start if we're at the end. We have mapped the 'I' key to PianoMachine.changeInstrument; use this method in your implementation.

a. [5 points] Write the spec for the changeInstrument method, and for any helper methods you expect to need.

b. [5 points] Write test cases for these methods.

c. [10 points] Add state to the PianoMachine class which reflects the instrument mode, fill in the changeInstrument method, and update any other code necessary so that your piano can switch instruments as specified. You may find the Instrument enum in the package 'midi' to be useful.

Problem 3: Switching Octaves

Pressing the 'C' and 'V' keys should shift the notes that the keys play down and up, respectively, by one octave (12 semitones). We should be able to shift by two octaves, maximum, in either direction from the starting pitches. We have mapped the appropriate keys to PianoMachine.shiftUp and PianoMachine.shiftDown; use these methods in your solution.

a. [5 points] Write the spec for the shiftUp and shiftDown methods, and for any helper methods you expect to need.

b. [5 points] Write test cases for these methods.

c. [10 points] Add this functionality by adding the appropriate state to PianoMachine, implementing the shiftUp and shiftDown methods, and updating any other methods necessary.

Problem 4: Recording and Playback

The piano should have the ability to record and playback sequences of notes, preserving the rhythm they were played with. 'R' should toggle record mode on and off, and 'P' should trigger playback. As before, we've provided you with signatures for PianoMachine.record and PianoMachine.playback as entry points; use these methods.

a. [5 points] Write the spec for the record and playback methods, and for any helper methods you expect to need.

b. [5 points] Write test cases for these methods.

c. [15 points] Add this functionality, by making PianoMachine keep track of the necessary state, implementing the playback and record methods, and adding any other necessary code. You might find the NoteEvent class useful.

Note: try triggering some new notes while the piano is in playback mode. You may find that the notes all sound together after playback finishes, as a single chord. Consider why this might be the case. For this problem, we will not deduct points for this behavior.

Problem 5: Keyboard Input During Playback

[10 points] If your application exhibits the (undesirable) behavior we describe at the end of Problem 4c, eliminate that behavior to complete this problem. You may edit piano.PianoApplet for this part of the assignment.

Hint: You may find the method KeyEvent.getWhen() to be useful.

Optional Extension

This extension is optional and will not affect your grade in any way. The only reason to try it is to quench your burning desire to program.

Add a "chord mode" to your keyboard, where each key you press will trigger several notes at once. A straightforward mapping would be to trigger a major triad for each note (e.g. pitches C, E, and G, if you press '1'.) You may also wish you make a different harmonization, at your musical discretion.

Before You're Done...

Double check that you didn't change the signatures of any of the code we gave you.

Make sure the code you implemented doesn't print anything to System.out. It's a helpful debugging feature, but writing output is a definite side effect of methods.

Make sure you don't have any outdated comments in the code you turn in. In particular, get rid of blocks of code that you may have commented out when doing the pset, and get rid of any TODO comments that are no longer TODOs.

Make sure your code compiles, and all the methods you've implemented pass all the tests that you added.

Does your code compile without warnings? In particular, you should have no unused variables, and no unneeded imports.

Make sure you check the last version of your code in SVN is the version you want to be graded.

Try to make sure that your code conforms to standard Java naming conventions. That is, class names should be StartingUpperCamelCase, and variables and methods should be startingLowerCamelCase.

6.005 PS1

1 pager instructions

MIDI keyboards

Specs new

- hard to write a spec for something I didn't write

But how are we supposed to implement w/o info!?!

Need to build pitch mapping

- No to Midi Tree
- then this is too easy ...

Insts for channel

or stop notes

think edit channel

They don't say - can have it force all notes to end

I could keep track of that

Too many things to keep track of in this class

- multiple ways to critique

② No history already there
Sucks can't modify piano

Now can do stop Notes, stop All Notes
— ? and test?

Do need spec for private methods?

Need iterator on Instrument

Oh timing does not test

How to get that instance?

Other people have methods

Can't instantiate instrument
— no I got it
— still don't fully understand
— I should work w/ CA

Actually I don't think I need my db of active notes
— review later

No need it for endNote

(3) Strings not matching!
- even to String!
Or type casting

Fixed - strings need compareTo!
- == is for same object

Part 3

Do we switch octives mid note - no
So just change mapping by 12
2 octives max!

No track octive in PM
Then track it when we play a notes

I should check no left over notes
to be really complete

Now have each instance

So stop stops all note instance
from all instruments
but current octive ← making weird design choices

(9)

Starting to get test driven dev

Still need to test for left over notes.

Record where?

? Use existing history

? How stop recording?

- play?

Oh toggle

Why is preRecord not saving?

Oh really stupid mistake

- 3:20AM

9/22 What type is recorded file ?

Grrr - ~~something~~ get history fails when no history

-fix online

Should ~~there be~~ recorded file not be a string?

- no its a string
- split by space
- need parser

(5)

Octive complicates

No just play those packs

Stupid, can't edit original file

So need to detect instument / pitch chanes

unless I save them in my own history file

Yeah prob want to do custom logger

Oh wait is in diff file

Need to make one

Hard to test recording this way!

Need to clean up loose ends on testing

- review old tests that don't match

Or move to new method

Lesson do whol sol before cleaning

⑥ Recitation

Actually playing tried
- works
- except play back
- they are there
- but prewriten records

So now it works
But plays all notes at once
Need to restore time
Should make log a fn

Ok working on doing this

(7) Evening W20

Now fix ~~data~~ timer

What is wrong w/ it?

milliseconds seem ~~[illegible]~~ right

1000 = 1 sec

So why taking so long to play back

? just first one?

Or extra 1000 added

-? from our mathh

just store long

? or multiple waits?

- don't think

Wait needs Int

Oh on + off

my key release not saved I think

note times out

holds key too long

(8) Wait seems to be waiting too long

Ohh midi wait in hundredths!

fixed

Oh 6.111 has moved to 32-044

Oh well I think I can figure out

Verify all tests

Why is there a extra 10 sec pause in there?

Tests all pass

— could make more elegant if had more time

Now problem 5

Eliminate by not playing

But can't edit piano applet

Lots of people stuck - give up + study 6.004?

I'll go ahead + edit PianoApplet

Give up for now

Returnins

Returnin allows you to revise and resubmit an individual problem set in order to try for a higher grade. Here's how it works:

- Just once.
  You can resubmit each problem set only once.
- Up to half the points back.
  Your returnin will be graded the same way as your original submission. You can earn at most half the points back that you lost on the original. Thus your final grade for the problem set will be max(originalGrade, (originalGrade + returninGrade)/2).
- Must use code reviews.
  The returnin must address all the code reviews received by the original submission, either by changing the code to reflect the review or by including a comment responding to the review and explaining why the code wasn't changed. A grader will check the returnin and will disallow it (without possibility of further returnin for that problem set) if it hasn't addressed the code reviews.
- Must show prior effort.
  Returnin is intended as an opportunity to reflect on and correct mistakes, not as a way to get an extension on the original deadline. If the grader judges that the original submission didn't show effort across the entire problem set, then the returnin will be disallowed.
- Returnin to your TA.
  To submit a returnin, make sure your code is committed to Subversion, and then email your recitation TA telling them which problem set you're resubmitting.
- Due before team projects.
  It's a good idea to do your returnin as soon as possible, while the problem set is still fresh in your mind. If you can't do it immediately, though, there are final deadlines. Returnins for PS0-PS4 are due when Project 1 starts, and returnins for PS5-PS7 are due when Project 2 starts. The reason for this is fairness to your teammates, who need you to put your energy into the project, rather than into returnin. The exact returnin deadlines are listed on the course calendar, and are generally the first team work day of each project, to give time for the last problem set to be graded and returned, and then for you to revise it if necessary.
- No slack.
  Slack days can't be used for the returnin deadline.
- Collaboration policy.
  Since returnin inevitably happens after code review for the problem set, it's understood that you've looked at other students' written solutions, and been inspired by other ways to solve the problems. You must be exceedingly scrupulous, therefore, in not using those written solutions during your revision. Both your original code and your revised code must be your own. Looking at other students' answers to the problem set while you are revising your solution will be considered a violation of the collaboration policy.

6.005 Lecture

9/26

Regular Expressions and Grammars

(Lexer + Types - like P-set)

(Type - one long enum)

- ☐ grammars
- ☐ regular expressions
- ☐ lexer + Parser

PS2 - State machine w/ piano

Not good w/ complex jobs

- like parsing

Markup languages

HTML	This is <i> italic </i> text
Markdown	This is _italic_ text
LaTeX	This is {\em italic and {\em not} text.}

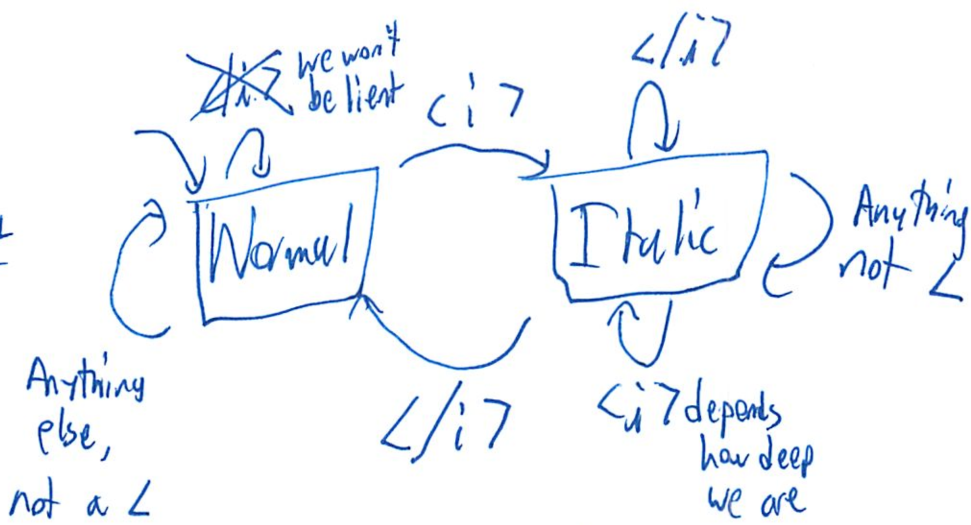
↑ can actually nest

HTML won't nest italic

Markdown can't nest; need diff start + end text

(2)

HTML



need to escape characters when actually want symbol "quoting" "escaping"

hard to clarify in notation

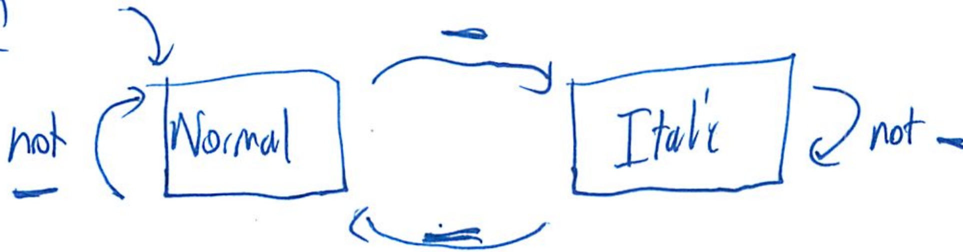
could draw

N <i> <i> ... <i>

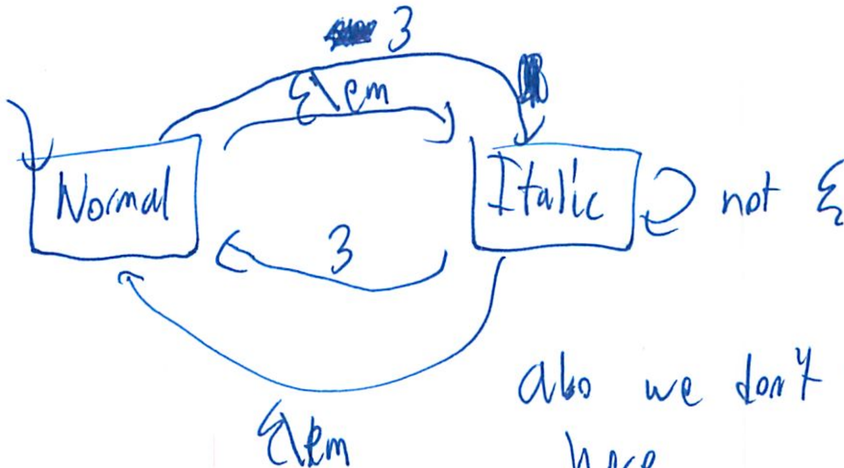
So in HTML < ← we will only prohibit < since start of tag. But HTML also prohibits > (And we could just look for whole <i>)

now need &

Markdown



Latex



also we don't show depth here

③

<u>Grammars</u> mirror sentence structure

Set of rewrite rules (productions)

Markdown ::= (Normal | Italic)*

- ↑ Markdown: nonterminal
- ↑ ::= : say "grammar"
- ↑ | : or
- ↙ Normal, Italic: other nonterminals or terminals
- ↑ * : any # of them repeated 0 or more times

Italic ::= _ Text _

- ↑ first _ : nonterminal
- ↑ between _ and Text: sequencing
- ↑ last _ : terminal

Normal ::= Text

Text ::= (A|B|C|D .... |a|b| ... & | < | ... ß)*

- ↑ but not _
- but Unicode has 65,000 of these!

[ABC ... abc ...]* or [^ _]*

- ↑ complement set designated by ^

HTML ::= (Normal | Italic)*

Italic ::= <i> Text Italic* Text </i>

- Italic*: might be 0 or more Italics inside

Text ::= [^ <]*

- ↑ any chars except [

(4)

So what can match italic?

~~<i> abc def </i>~~

<i> abc def </i>

<i> abc <i> c </i> d </i>

<i> a <i> b </i> c <i> d </i> </i>

So that defn is too restrictive

Italic ::= <i> (Text | Italic)* </i>

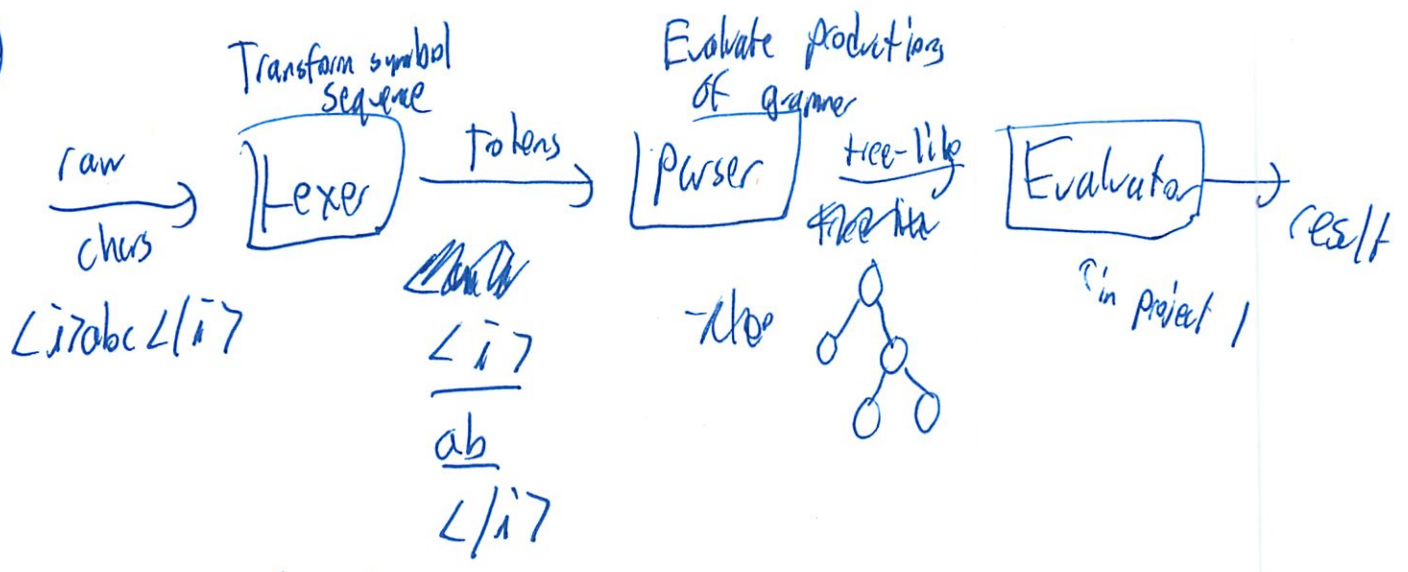
Grammar is ambiguous - can write multiple ways
- not bad for design
- when implementing, must decide how you want it to go
- if auto building parser, often requires unambiguous

PS3: split up into 2 parts

Lexer
- turn raw char stream
- produce seq of tokens

Parser
- look at relationships
- evaluates productions

(5)

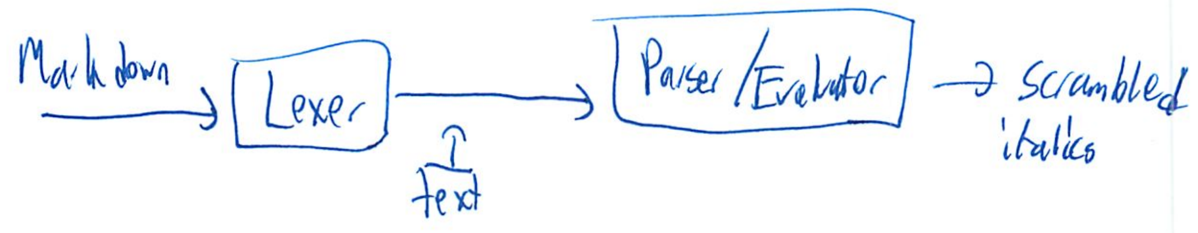


- often throws away unneeded chars
- removes white space

-

PS3 | Parser + Evaluator in 1

Now build Markdown parser in class



1st need representation for token
- underline or text? ← type
- actual value ← value

List all types in enum

6

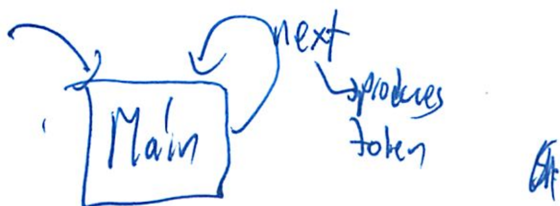
List possible values in Type Enum

Bad design to mix language together
- if only 1 lang can have 1 enum

By seperating jobs each part has clear responsibility
But not really all that seperate
like can't plug a diff parser in
But easier to debug

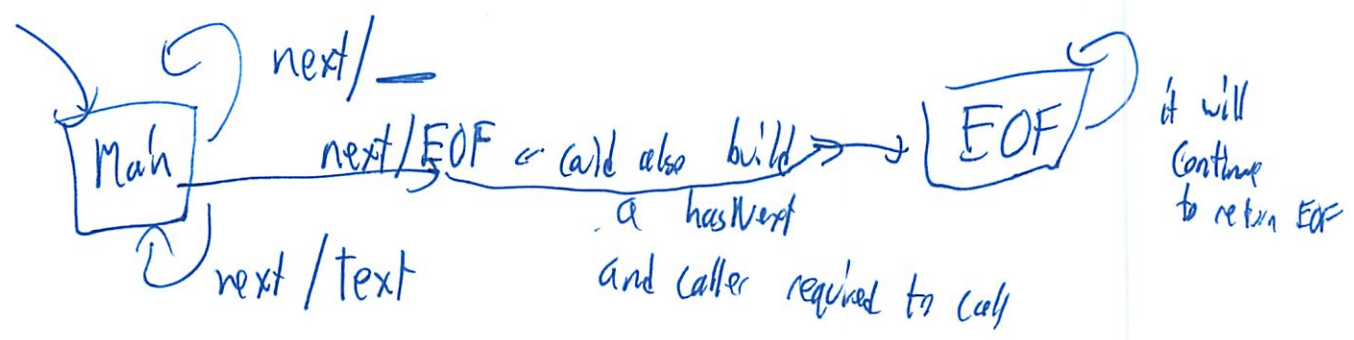
Token
- has type
- has value

Markdown Lexer
- SM
- give raw char to start
- has 1 state transition next()



(1)

Draw as



So next has switch of diff chars
returns tokens (UNDERLINE, value)
for each part

Then Parser
- creates a lexer

Eval method
- parse + evaluate

(This one scrambles letter)

Gets tokens from lexer
- get type
- switch on types
- returns values evaluated properly
- have evaluate method
- that does the shuffling

L5: Regular Expressions & Grammars

Today

- Lexing and parsing
- Grammars
- Regular expressions

Required reading (from the Java Tutorial)

- Enums
- Regular Expressions

Markup

For today's working examples, we'll be using several different *markup languages*, which represent typographic style in plain text. Here they are:

HTML

```
Here is an <i>italic</i> word.
```

Markdown

```
This is _italic_.
```

LaTeX

```
In LaTeX, {\em italics are used to show {\em emphasis}, unless you're
nesting emphasized text inside other emphasized text}.
```

State machine review

Let's start by drawing state machines representing the behavior of a renderer for these markup languages. All three have two states:

Normal Italic

What differs are the transitions between them.

The state machines alone are dissatisfying – they don't tell the whole story about the language. In particular, they don't show the *structure* of the language, particularly of LaTeX.

Reading Input

We're going to build some classes that read and interpret these markup languages.

The first thing we need to do is decide on the set of events we want our state machine to use. Strings and streams give us very fine-grained events, like characters, but the kinds of input we want to process usually have bigger symbols than that. So it will be useful to divide input into two steps:

These lecture notes have been collaboratively authored, with contributions from Saman Amarasinghe, Srini Devadas, Michael Ernst, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. *Copyright © MIT.*

- lexical analysis, or *lexing*, which transforms the stream of characters into a stream of higher-level symbols, like words, or HTML tags, or whole chunks of text. These symbols are usually called *lexemes* or *tokens*.
- *parsing*, which takes the stream of tokens and interprets them. The parser is responsible for knowing the relationships between them (e.g., checking that <i> precedes </i>).

In typical practice, these two steps are designed as independent state machines, a lexer and a parser, that interact through a clean interface: the output events of the lexer are consumed by the parser. This is an instance of a general software design principle called *separation of concerns*: the lexer worries about lexing (e.g., what an individual HTML tag like "<i>" should look like), and the parser worries about parsing (e.g. that "<i>" should precede "</i>"). Although they are closely coupled in the sense that you can't use the parser without the lexer, they still have a clear contract between them.

Lexical Analysis

Lexical analysis takes a stream of fine-grained, low-level symbols (e.g., characters) and aggregates them into a sequence of higher-level symbols (e.g. words), called lexemes. The process is also called *tokenization*, and its output symbols are *tokens*.

Tokenization makes the second stage of input handling (parsing) simpler, by abstracting out some of the details of the input. For example, a lexer for Java throws away information that the compiler doesn't care about, like whitespace and comments.

```
/** square a number */
int square (int x) {
   return x*x;
}
```

might produce a token sequence like:

```
int   square   (   int   x   )   {   return   x   *   x   ;   }
```

It can also combine symbols into classes that are useful to the parser. For Java, for example, user-defined names are typically grouped into a single kind of token, *identifier* or *id* for short, that also carries along information about the particular name:

```
int   Id("square")   (   int   Id("x")   )   {   return   Id("x")   *   Id("x")   ;   }
```

So tokens are not just strings, but might be objects with fields. In Java, an *enum* class is a useful way to define tokens.

Different languages call for different kinds of tokenization. In Python, for example, you wouldn't throw away all whitespace entirely; you'd have tokens for newlines and tokens for indentation, since those affect Python statement structure. In natural language processing (like English), a tokenizer might detect parts of speech (nouns, adjectives) and undo morphology (e.g. "mice" becomes "mouse+plural").

For our markup languages, we certainly want tokens for the italic syntax (_ , <i>, {\em, etc.). We might also considering throwing away whitespace, but let's not; whitespace is actually significant in these formats (Latex and markdown pay attention to blank lines, for example). So instead we'll just treat all other text as a single kind of token called *text,* like this:

```
HTML
This is an <i>italic</i> word.

Text("This is an ")   <i>   Text("italic")   </i>   Text(" word.")
```

Markdown

```
Words are _italic_.

Text("Words are ")    _    Text("italic")    _    Text(".")
```

LaTeX

```
You can {\em nest {\em italics} in Latex}

Text("You can ")    {\em    Text(" nest ")  {\em   Text("italics")  }
Text(" in Latex")   }
```

Lexer

A lexer is a state machine that does lexing. Like an iterator, a lexer typically has one method *next()* that returns the next token in the sequence. Inside the lexer is a state machine that processes the characters of the input stream in order to generate the token sequence.

You also have to make a design decision about how the lexer signals the end of the sequence. One option is the approach taken by Iterator: a method *hasNext()* that indicates whether another token is available. Another option is a special END token; InputStreams use this technique when read() returns -1. Another option is throwing an exception from *next()*. Some of these alternatives are discussed in the lecture on Specifications.

Grammar

- a grammar defines a set of sentences
- a sentence is a sequence of symbols (tokens, also called *terminals*)

- a grammar is a set of productions
- each production defines a non-terminal
- a non-terminal is a variable that stands for a set of sentences

By convention, nonterminals are capitalized, and terminals are lowercase.

production has form

- non-terminal ::= expression of terminals and non-terminals and operators

The three operators are:

- sequence: A ::= B C   an A is a B followed by a C
- iteration: A ::= B*   an A is zero or more B's
- choice: A ::= B | C   an A is a B or a C

You can also use additional operators which are just syntactic sugar (equivalent to combinations of the big three operators):

- option: A ::= B? an A is a B or is empty

  grouping: A ::= (B C)* parentheses for grouping an A is zero or more B-C pairs

- 1+iteration: A ::= B+ is equivalent to A ::= BB* an A is one or more B's

  character classes: A::= [abc] is equivalent to A ::= a | b | c

  A::= [^b] is equivalent to A ::= a | c | d | e | f | ... (all other characters)

example:

grammar

URL ::= Protocol :// Address

Address ::= Domain . TLD

Protocol ::= http | ftp

Domain ::= mit | apple | pbs

TLD ::= com | edu | org

terminals are

://, ., http, ftp, mit, apple, pbs, com, edu, org

non-terminals are

TLD = { com, edu, org }

Domain = { mit, apple, pbs }

Protocol = { http, ftp }

Address = { mit.com, mit.edu, mit.org, apple.com, apple.edu, apple.org, pbs.com, pbs.edu, pbs.org}

URL = { http://mit.com, http://mit.edu, ..., ftp://mit.com, ...}

Here's the grammar for our simplified version of markdown:

Markdown ::= (Normal | Italic) *

Italic ::= _ Text _

Normal ::= Text

Text ::= [^ _]*

Here's the grammar for our simplified version of HTML, which allows italic regions to be nested inside other italic regions:

Html ::= (Normal | Italic) *

Italic ::= <i> Html </i>

Normal ::= Text

Text ::= [^ _]*

And here is our Latex grammar:

Latex ::= (Normal | Italic) *

Italic ::= {\em Latex }

Normal ::= Text

Text ::= [^ _]*

Regular Grammars

A *regular* grammar has a special property: by substituting every nonterminal (except the root one) with its righthand side, you can reduce it down to a single production for the root, with only terminals and operators on the right-hand side. This "compiled" form of a regular grammar is called a *regular expression.*

The markdown grammar is regular. By replacing nonterminals with their productions, it can be reduced to a single nonrecursive production:

Markdown ::= ([^ _]* | _ [^ _]* _)*

The expression on the righthand side, consisting only of terminals and operators, is called a **regular expression**. It's far less readable than the grammar, but it's fast to implement, and there are many libraries in many programming languages that support regular expressions (called *regexes* for short). More on this later in the lecture.

A grammar that can't be reduced to a single nonrecursive production is called *context-free*. Both the HTML and Latex grammars are context-free. The grammars for most programming languages are also context-free. In general, any language with nested structure (like nesting parentheses or braces) is context-free. Here's part of the grammar for Java statements:

Statement::= Block

| if ParExpression Statement [else Statement]

| for (ForInit? ; Expression? ; ForUpdate?) Statement

| while (Expression) Statement

| do Statement while (Expression) ;

| try Block (Catches | Catches? finally Block)

| switch (Expression) { SwitchBlockStatementGroups }

| synchronized ParExpression Block

| return Expression? ;

| throw Expression ;

| break Identifier? ;

| continue Identifier? ;

| ExpressionStatement

| Identifier : Statement

| ;

Grammars and State Machines

regular grammars vs state machines

- a state machine's trace set is prefix closed: if t^e is a trace, so is t
- regular grammars can express trace sets that are not prefix closed

traces of (up down)* include <up, down> but not <up>

- so grammars are more expressive

but can add "final" states to state diagrams

- then define (full) traces as those that go from initial to final states
- now regular grammars and machines are equally expressive
- they both define regular languages

in practice

- use state machines for non-terminating systems
- use grammars for terminating and non-terminating systems

Recursive descent parsing and evaluation

The grammar guides the design of your parser class. The code below shows an example of a recursive-descent parser for the markdown grammar.

```
/**
 * A Gallileo object is a parser/evaluator for markdown that scrambles
 * italic text (generates a random anagram of each italic part) so that Kepler
can't read it.
 */
public class Gallileo {

    private final MarkdownLexer lex;

    public Gallileo(String markdown) {
        this.lex = new MarkdownLexer(markdown);
    }

    /**
     * Evaluate the input text, scrambling italic sections.
     * Can be called only once on a given object.
     * Modifies this object, consuming all the text.
     * @return string of text with markdown formatting removed
     * and italic sections replaced by a random anagram.
     * For example, new Gallileo("The killer was _Mrs. White_").eval()
     * ==> "The killer was hrW.sM tie"
     */
    public String eval() {
        return evalMarkdown();
    }
```

```
    // Grammar:
    //   Markdown ::= (Normal | Italic)*
    //   Normal ::= Text
    //   Italic ::= _ Text? _
    //
    //  (Text and _ are tokens generated by MarkdownLexer)

    /**
     * Evaluates the Markdown production of the grammar.
     * Modifies lex by consuming all the remaining tokens.
     * @return evaluated string
     */
    private String evalMarkdown() {
        StringBuilder sb = new StringBuilder();

        for (Token tok = lex.next(); tok.getType() != Type.EOF; tok =
lex.next()) {
            switch (tok.getType()) {
            case UNDERLINE:
                sb.append(evalItalic(tok));
                break;
            case TEXT:
                sb.append(evalNormal(tok));
                break;
            default:
                throw new AssertionError("unexpected token: " +
tok.getType());
            }
        }

        return sb.toString();
    }

    /**
     * Evaluates the Normal production of the grammar.
     * Modifies lex by consuming an entire production, including the last token
of the production.
     * @param tok Token that started this production (required to be TEXT)
     * @return evaluated string
     */
    private String evalNormal(Token tok) {
        // normal text isn't changed by this process, just return it as-is
        return tok.getValue();
    }

    /**
     * Evaluates the Italic production of the grammar.
     * Modifies lex by consuming an entire Italic production, including its
final token.
     * @param tok Token that started this production (required to be UNDERLINE)
     * @return evaluated string
     */
    private String evalItalic(Token tok) {
```

```
        StringBuilder sb = new StringBuilder();

        // the passed in tok is UNDERLINE; skip it and advance to the next

        // note that this code actually evaluates _ TEXT* _, not just _ TEXT? _
        for (tok = lex.next(); tok.getType() != Type.EOF && tok.getType() !=
Type.UNDERLINE; tok = lex.next()) {
            if (tok.getType() == Type.TEXT) {
                // collect and shuffle the text
                sb.append(shuffle(tok.getValue()));
            } else {
                throw new AssertionError("unexpected token: " +
tok.getType());
            }
        }

        return sb.toString();
    }

    /**
     * Make a random anagram of a string.
     * @param s string to rearrange
     * @return a random permutation of the characters in s.
     * For example, shuffle("abc") might return "bca" or "cba" or "abc".
     */
    static String shuffle(String s) {
        // this is not the best way to implement this -- what would be better?

        // split with empty-string separator to get each char as a string
        String[] a = s.split(""); // e.g. "", "a", "b", "c"    (produces an
extra empty string, but that won't hurt)

        List<String> l = Arrays.asList(a);
        Collections.shuffle(l);   // now it's shuffled, e.g. "a", "", "c", "b"

        // glue the shuffled list back together into one string
        StringBuilder sb = new StringBuilder();
        for (String t : l) {
            sb.append(t);
        }

        return sb.toString();
    }


    /**
     * Main method.
     */
    public static void main(String[] args) {
        Gallileo g = new Gallileo("I've discovered that _Saturn has ears_.
Suck it, Kepler!");
        String message = g.eval();
        System.out.println(message);
```

```
    }
}
```

Parser generators

For some grammars, particularly more complex context-free grammars, you need heavier machinery. *Parser generators* are a good tool that you should make part of your toolbox. A parser generator takes a grammar as input and automatically generates parser code for that grammar – typically both a lexer and a parser. JavaCC is a mature and widely-used parser generator for Java.

Using regular expressions

Regular expressions (“regexes”) are even more widely used in programming tools than parser generators, and you should have them in your toolbox too.

In Java, you can use regexes for manipulating strings (see String.split, String.match, java.util.regex.Pattern). They’re built-in as a first-class feature of modern scripting languages like Perl, Python, Ruby, and Javascript, and you can use them in many text editors for find and replace. Regular expressions are your friend! Most of the time. Here are some examples:

replace all runs of whitespace with a single space, strip leading and trailing spaces:

```
string.replace(“\s+”, “ “).replace(“^\s+”, “”).replace(“\s+$”, “”);
```

extract part of an HTML tag

```
Matcher m = Pattern.compile(“<a href=’([^’]*)’>”).matcher(string);
if (m.matches()) {
        m.group(1) is the desired URL
}
```

Risks of regular expressions

safe from bugs?

easy to understand?

ready for change?

Summary

6.005 Elements of Software Construction | Fall 2011
Problem Set 3: Calculator Parser
Due: Thursday, September 29 2011, 11:59 PM

This problem set explores parsing.

Much of the code that you write for problems in this problem set will depend heavily on how you decided to implement earlier parts of the problem set, so you may want to read through the entire assignment before jumping into writing code.

Do not change the signatures or specifications of any methods, classes, or packages that we have provided you. Your code will be tested automatically, and will break our testing suite if you do so.

To get started, pull out the problem set code from SVN admin.

Overview

It's often convenient to use different units in the same computation. For example, to figure out how many lines of twelve-point type fit in a six-inch column, it would be nice to have a calculator that accepted the expression "6in/12pt". Construct two grammars for such a language: first, a lexical grammar that breaks the sequence of characters into numbers, unit specifiers, operators, and left/right parentheses, filtering out spaces and tabs (but not requiring them as delimiters); and second, a syntactic grammar that groups expressions appropriately.

Your grammar should be able to recognize the following types of expressions:

- Arithmetic expressions with +, -, *, and /. You can assume that the order of operations will *always* be made explicit with parentheses.
- Expressions with integer and decimal operands, both as scalars and inches and points. Point operands will be the number followed by 'pt,' and inches will be followed by 'in.' The result of evaluating the expression should be in appropriate units also.
  - inches/scalar = inches
  - inches*scalar = inches
  - inches/inches = scalar
  - inches/points = scalar

  Assume similar combinations for substitutions of inches and points.

  The following combinations are not as intuitive, so we've supplied this specification to allow for consistency in grading:

  - scalar/inches = inches
  - inches*inches = inches
  - scalar+inches = inches
  - inches+points = inches (use units of the first operand)
- Unit conversion expressions (also made explicit by parentheses), represented by 'in' or 'pt' following an expression.
- As said above, whitespace should be ignored.

You will be expected to be able evaluate all the following expressions, with the result presented after the '=':

- 3+2.4 = 5.4
- 3 + 2.4 = 5.4
- (3 + 4)*2.4 = 16.8
- 3in * 2.4 = 7.2in
- 4pt+(3in*2.4) = 522.4pt
- (3 + 2.4) in = 5.4in
- (3in * 2.4) pt = 518.4pt

test cases

You will NOT be expected to evaluate the following expressions:

- 3+2+1 (order of operations not made explicit)
- 2+(3+4)+2 (order still not explicit)
- (3+4)^2 (do not need to support ^)
- 2+3in pt (order not explicit)

If your calculator receives an expression that it does not know how to handle, you should display some sort of error message, ideally with some information about why the input could not be evaluated, such as "Order of operations not made explicit," or "Cannot divide by 0."

We have provided you with some skeleton code for this calculator. It consists of a simple prompt loop, in the main method of MultiUnitCalculator, for users to enter expressions. There is also a Lexer class along with a Type class, and a Parser class for interpreting the input expression String and evaluating it. This problem set will walk you through the implementation of these classes.

Problem 1: Specify a Grammar

Before jumping into the implementation, you should specify a grammar for your calculator. Your grammar should be able to recognize the expressions described in the **Overview** section above.

a. [5 points] Start by deciding what symbols or terminals your grammar will contain. This should include things like '+,' '6,' and 'pt.' List these in a comment at the top of Type.java.

b. [5 points] Next, group your symbols into Types that your Lexer will recognize, based on their function in an expression. For example, plus, divide, and open parenthese will be different Types, but you can specify just one Type for all numbers. Indicate your groupings in your comment in Type.java and then create a Type enum for each of your groups.

c. [15 points] Finally, write out your grammar in a comment at the top of Parser.java. Use the production syntax defined in lecture.

Problem 2: Implement the Lexer

The first step for evaluating an expression will be converting the user input String into a set of tokens that your parser will be able to recognize. These tokens will be parts of the String paired with the Types that you defined in Problem 1. We've provided you with Lexer.java, which defines an internal Token class for you to utilize when implementing your Lexer. To actual recognize the tokens in the input Strings, you may want to use regular expressions, as discussed in Lexer. Java provides support for regexes in the Java.util.regex package, as presented in lecture. Eventually you will pass the Lexer to the Parser constructor, but exactly how the Lexer presents the Tokens to the Parser is up to you. *Hint: You can think about the Lexer consuming the expression one character at a time and passing the resulting Tokens to the Parser, but this is only one way of doing things.*

a. [5 points] Write the spec for the Lexer constructor method, and for any other methods you expect to need for tokenization.

b. [5 points] Write test cases for these methods in a separate file.

c. [10 points] Now implement your Lexer class based on your specification. Test it using the tests that you wrote.

Problem 3: Implement the Parser

Your parser should use the tokenization produced by the Lexer to parse the expression according to the grammar you defined in Problem 1. Exactly how you do this is up to you; the only thing that we enforce is that the Lexer is passed as an argument to the Parser constructor.

a. [10 points] Write the spec for the Parser constructor method, the evaluate method, and any other methods you expect to need for parsing and evaluating the tokenized expression.

b. [5 points] Write test cases for these methods in a separate file.

c. [20 points] Implement the Parser and make sure all your tests pass.

Problem 4: Put it all together

Finally, apply your Lexer and Parser to the input expression as defined in the spec for evaluate in MultiUnitCalculator.java.

a. [5 points] First, write test cases for the evaluate method in CalculatorTest.java.

c. [10 points] Implement the evaluate method using your Lexer and Parser. *Hint: This shouldn't be too hard, or you might want to rethink how you wrote your Lexer and Parser.*

Optional Extension

This extension is optional and will not affect your grade in any way. The only reason to try it is to quench your burning desire to program.

Play around some more with unit conversion. Add support for different units, like seconds or mph.

You might also want to experiment with evaluating expressions where the order of operations is not made explicit by parentheses, and your parser must resort to PEMDAS!

Before You're Done...

Double check that you didn't change the signatures of any of the code we gave you.

Make sure the code you implemented doesn't print anything to System.out. It's a helpful debugging feature, but writing output is a definite side effect of methods.

Make sure you don't have any outdated comments in the code you turn in. In particular, get rid of blocks of code that you may have commented out when doing the pset, and get rid of any TODO comments that are no longer TODOs.

Make sure your code compiles, and all the methods you've implemented pass all the tests that you added.

Does your code compile without warnings? In particular, you should have no unused variables, and no unneeded imports.

Make sure you check the last version of your code in SVN is the version you want to be graded.

Try to make sure that your code conforms to standard Java naming conventions. That is, class names should be StartingUpperCamelCase, and variables and methods should be startingLowerCamelCase.

Doing 6.005 PS3

9/25

Ugg Parser does not sound fun
But I like how they break it up for us

Type.Java

So add as comments?
Then what format in types
So + / are different types?
I have no clue what this file is supposed to look like
Perhaps continue for now

Lexer

What should it return?
What are these tokens again?
Look at parser
Perhaps wait for class

Mon - Regex + Grammars
Wed - Abstract data types

②

Its not that I don't get those things

- I did something like this in Python
- Its how they want us to do this

Like if I saw an example

Or got help

Like if they wrote specs

~~[illegible]~~ Or if I was starting from scratch

I may ~~have~~ also just try something

May be scared of doing OO correct

Go to LA hrs tomorrow

8/26 Lab

Ok lecture cleared things up

Implement what I remember ~~[illegible]~~ for it as

- Code not posted yet

~~[illegible]~~ What do do w/ parenthese

- just code in Lexer (w/ null values?)
- then parse?

(3)

How access the type?

Type.START_PARENTESIS

¿Then internal list of tokens?

Code posted!

Oh can instantiate w/ 1 param

Just return Token on next

Java does thing where ~~for~~ for global variables need to instantiate in class def.

Then can just call it (no new)

or this.

Can also initialize/set in constructor

do string - not ~~char~~ array so can do multiple chars in a switch

---

~~K~~nowing what you are doing makes you feel in control - like a real engineer

Actually just copying off not as fun
- but I am doing things a bit different

I like auto complete

Now I should write some tests for it

'' = character
" " = string

finally getting good w/ Java

Stuck on char printing - why does this time out!
My while loop - is Java reading all of it?
- no it should not be
- Oh order of opps?
That was it - thought did Ors first
- oh well - took 30 min to find + fix

Now need to do list equals w/ Junit

(5)

~~Refen~~
Can build compairs helper function

I am finding typing from scratch - not comparing

(✓) Woot passed test

Wrote my own testing system in 11 min!.

Write more tests

- EOF not working - hmm

Fixed

Oh can ~~only~~ run only 1 test at a time!

Committed at :33 :44 :55 :)

Perhaps too much testing

But doing all in book

Other students don't like test cases

Benefits of abstraction :)

Oh done

(6)

Todo: Write Grammer in parser

Now write parser method

- Copy of code in class

Note here that lexer is passed in

- Oh ~~was~~ no need to save
- is auto done in Constructor, if vairable names same?
  - No think need this.lexer = lexer

Here we are trying to do Math

: Order of ops

Since we are not going straight across

Oh Value type

How to do the nesting R parenthesis?

So evaluate L → R
- when see parentheses call evaluate on that?

Nice for loop from prof.
- neat did not know could do for loop on objects

(7)

Need to do order

(Float Operand Float)

Parentese inforce

So if Parentuses → keep pulling then send eval

w/ those tokenss

Still figuring out - might not be very neat

Oh darn Value is of that proper type

- no need for Object [ ]

I should test soon

- basic structure

Or perhaps have dinner

I think I get what they want

- no need for TA

- have not asked 1 qu

Now need to start tests

- dinner break

⑧

9/26
Home

Start putting together tests
Having trouble getting type enum
- here is a real Java question
Emailed in

What does making a class static do again?
- Worked w/ token
- makes "nested - top level" classes
- just like main class
- emailed in

So test now returns 0 - which sounds right
Find out/Fix main class now
So now I can eval (2 + 3.4) but not 2 + 3.4?

Opps need break; after every case in switch()
Got it to work w/ ()

(9)

Ok now how to get to work w/o parentesis

- basically just eval
- perhaps pull all tokens
- Then do parentesis
- or just evaluate whole thing

- I should be able to do

$$5 \cdot (7 + 10)$$

- but units mucks it up - should be flags
  - not tokens
- So evaluate should see 5 • (unit)
- Then do () unit
- Question is how to match
- Could also do reverse and find first (on outside L → R

  and first R → L)
- But that not right for (5 + 7) • (8 - 2)

(10)

Would be great! If () was an object

with proper recursion

Need like a SM

- look at code

So eval underline starts on (

Then goes till underline

And appends

No recursion supported

This is a common problem in CS

- Can do top down or bottom up

Ok saw it

(= start → call yourself on rest of string

) = return

So try that

Or track depth somehow

(Think did in 6.01) - go check

- hard to search labs!

HW Assign 1 -- call recursive parse left, right close

(11)

Think have idea

- move () code to evalExpression
  - always call that

I always have to pause + think about recursion
because returns answers weirdly
esp here where types are separate tokens

[I enjoyed this assignment much more in bed]

Watch how lists are copied!

Cool ~~start~~ works - basic works
~~But lend~~
- but w/ parenteses keeps starting
  - need i+1!

Works!

Now just scale up tests
- do all non unit tests

Multiply did not work!
And multiple ()
No ~~() and ...~~ (3+4) · 2 · 4

(12)

No expected is wrong - value ends up is correct!
I just messed up in my description

No it was ~~mixe~~ mixed on test
Forgot to change symbol when actually doing math

Now need to fix parentheses issue
Then types
~~Break~~

On end - break out of for loop
- can make sep fn it can return from

Could easily extend this to do

$1 + 2 + 3$

So it now returns 7

$(3 + 4) \cdot 2.4$

↑ how to tell it to keep going

So I should be doing it right

(13)

Prob not modifying collection propely

And i is all wrong

- needs to be at end

So remove from collection always

Works! on 1st try :)

Thought that would be tricky

Now units

Oh unit conversion points to inches!

- did not do!

So w/ 1 minor tweak it works on

Units ~~conversion~~

But never did unit conversion

Points → Inches = 72

So look at our test cases

They didn't really explain in instructions

Can only go ~~In~~ pt → in or vica versa?

And • % only

(14)

Ok don't error
 - instead do conversion

inches · points

Also addition

inches + points = inches

↑ Converted to inches (/72)

Need to go through each rule

So $5 + \frac{72}{72}$

$\text{points} / \frac{\text{points}}{\text{inches}} = \text{inches}$

inches · points
 ↳ guess use 1st unit

Now need implicit conversion
Ok the plus as basic unit works!
7pt + 0 float = 72 pts

(15)

I think I could do more elegant
- but want to finish!

Inch → point
· 72

~~[illegible]~~

Ok done special case - but broke everything else!
Fixed!

So I think I am done

- reread P-set

~~[illegible]~~ So $\frac{\text{inches}}{\text{inches}}$ = ratio (scalar)

Test proper errors
- how get Junit
try ( )
Catch ( )
but no succeed ( )
- just skip
No! add as annotation!

(6)

Oh 5+5+5 does not fail nicely

Oh I can do w/ token

(Always throw assertion errors)
└ logically impossible error

Just do it now

Our need to write grammar
└ Only the input
- does not say output

Oh Part 4

So just lexer + parse

Copy from test case

Oh I already did all this!

Just copy test cases

Oh junit checks if returns true

So all true

Should be good

(17)

It should run though

Giving error

Try restarting Eclipse

Good now!

Woot it works

Just need to remove println

Ok!

But 5+7 in gives .1666

This is their code I think

Works in test!

Just the toString()

Oh really done now

6.005 Recitation

9/27

- Some changes + clarifications in P-set
  - make sure to support
- Most/all of my changes been verbally approved

Grammar

- Define set of strings valid based on grammar
- Non terminals
- Terminals
- Production Rules
- Syntax/Control characters

Book example (string of text

- ~~[illegible]~~
- Non terminals: Chapters, Book, Paragraph, Sentences, ~~[illegible]~~ Punctuation Title
- Terminals: Words, ~~[illegible]~~ " ", PB, ¶, Period, Exclaim, Question
- Production Rules

  Book ::= Title, Chapters+ (at least one)

  (^ PB)

②

Title ::= Word (" " Word)*

✓ technically not supposed to put terminals + non terminals

↑ no bar so must have one space than one word repeat this group as many times

Chapter ::=

→ ~~Paragraph~~

→ Title " " Paragraph (~~para break~~ ¶ Paragraph)*

+ 1 or more	
* 0 or more	? 0 or 1

Paragraph ::= Sentence (" " Sentance)*

Sentance ::= Word (" " Word)* ~~Period~~ Punctuation

Punctuation ::= ("!"|"?"|".")

Can redefine Sentance

Sentance ::= Word ((",", "-", etc) {0|1} " " Word)* Punctuation

also?

→ Need to ~~to~~ add to terminals

(3)

Things could specify

- Menu
- Expression
- Buildings
- Calendar
- Python

<u>Calendar</u> (Monthly flip calendar)

Calendar ::= Cover (br Month){12} br Final

Cover ::= Title Glossy_Pic

Month ::= Glossy_Pic Grid

Grid ::= Title (Weeks){4|5}

Weeks ::= (Cells){7}

Cells ::= Number (Holiday)?

Could even do

Number ::= Digit{1|2}

? but need 0-30?

(4)

So Number ::= [0-3] [0-9]

Could even do more

([0-2] [0-9]) | ([3] [0 1])

6.005 L06

Abstract Data Types

- [ ] ADTs
- [ ] rep independence
- [ ] rep exposure
- [ ] rep invariants
- [ ] abstraction functions

PS 3 due Thurs

This is the intellectual heart of 6.005
- This is not a Java class
- this is not a class on testing

Goals
1. Safe from bugs
2. Easy to understand
3. Ready for change

Words
- abstraction
- encapsilation
- modularity
- information hiding
- sep of concerns (Mon)

②

Abstract data types (ADT) is formalization of theses

- developed by Barbara Liskov (Turing Award)

type = Set of values with operations can perform on

int ① ⑤ ③ ⑨⓪⓪ +, -, *, /

<, >, .toString()

String.valueOf()

type turns into opaque types

- can <u>only</u> apply operations to
- details of whats going on inside hidden

Have <u>creators</u>

ADT T

Creator $t^* \to T$ ← can be multiple

producer $T^+, t^* \to T$ combining multiple instances

Observers $T^+, t^* \to t$ examine type in terms of other types we know

mutator $T^+, t^* \to \text{void} \mid t^*$ ← can return stuff

③

Can have static creator methods

└ called factory methods

(research more)

- here used to call value of()

No mutable methods here

Producer methods

- concat

Instance method always has "this" param

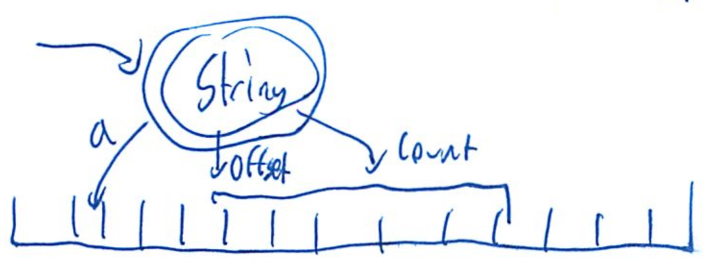
- don't see it in Java

(I am finally getting this)

The implementor can choose an internal representation of his/her choosing.

- want representation independence

Like String is actuall char array



So can return substrings cheaply

④

So this is why string must be immutable

List < > is not a class - its an interface
- just pure operations
- can only put sigs + specs
- no static methods either
- Need a class to implement
  - like ArrayList
  - needs to include all methods in abstract interface
  - Can have more than one
    - Linked List

Also want it to preserve its own invarients
↑ property of program that never changes
- an example immutability is crucial invarient of strings
- don't put mutator in ADT
- but also must make internal variables private
- can declare variable final
- but final on class means can't subclass

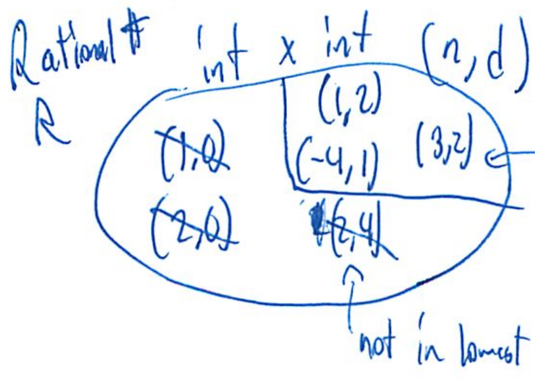
5

What about "public final"?
- that works - immutable
- but not representation independency

Calling final on a mutable object
- like Calendar
- does not work!
- it will always be pointing to that date object
- ~~which~~ date object can be mutated later
  - even passing same object in to new constructor bad
    - points to same object!
    - modifying - modifies underlying object
- so defensive copying or cloning when you return it in getDate()

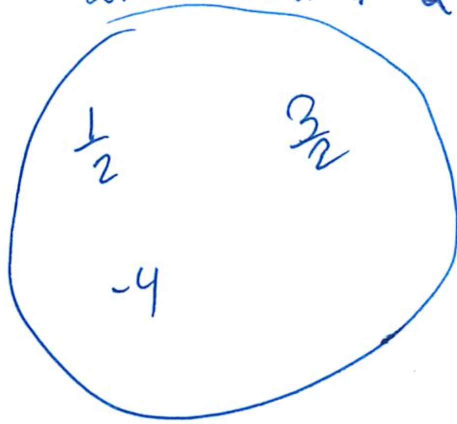
- Can have unmodifiableList
  - sets throw exception
  - but underlying list can still change

Math behind them

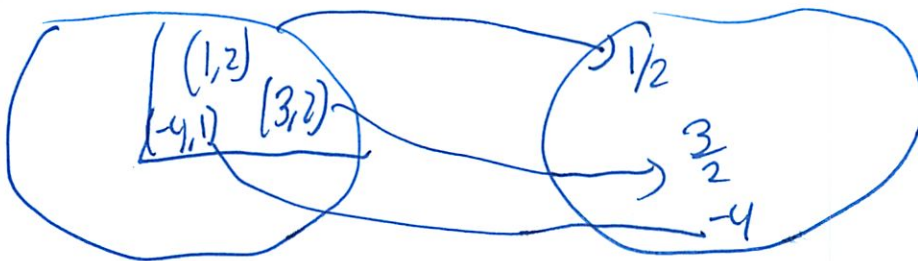


representational invariant
- denom ≠ 0
- numerator/denom must be in lowest terms

(6)



Map each of the representation value to abstract space value



If wanted to allow not lowest term

R space large

but now not 1:1 mapping

The implementor does ~~this~~ this

Does not show to client so don't lose rep invariance

Since invarient just means won't change while running

- can reprogram

(7)

Can do these checks in code
by building a check methods
Error if violates our invarient

Think about toString method
- makes it visual for humans

9/28

L6: Abstract Data Types

Today

- Abstract data types
- Representation independence
- Rep exposure
- Abstraction function & rep invariant

Required Reading (from the Java Tutorial)

- Interfaces
- Collection Interfaces (focus on Set, List, and Map)
- Collection Implementations (again, Set, List, and Map, but also Wrapper and Convenience)

In this lecture, we look at a powerful idea, abstract data types, which enable us to separate how we use a data structure in a program from the particular form of the data structure itself. Abstract data types address a particularly dangerous dependence, that of a client of a type on the type's representation. We'll see why this is dangerous and how it can be avoided. We'll also discuss the classification of operations, and some principles of good design for abstract data types.

User-Defined Types

In the early days of computing, a programming language came with built-in types (such as integers, booleans, strings, etc.) and built-in procedures, eg. for input and output. Users could define their own procedures: that's how large programs were built.

A major advance in software development was the idea of abstract types: that one could design a programming language to allow user-defined types too. This idea came out of the work of many researchers, notably Dahl (the inventor of the Simula language), Hoare (who developed many of the techniques we now use to reason about abstract types), Parnas (who coined the term *information hiding* and first articulated the idea of organizing program modules around the secrets they encapsulated), and here at MIT, Barbara Liskov and John Guttag, who did seminal work in the specification of abstract types, and in programming language support for them -- and developed 6170, the predecessor to 6.005. In 2010, Barbara Liskov earned the Turing Award, computer science's equivalent of the Nobel Prize, for her work on abstract types.

Objects

The key idea of data abstraction is that a type is characterized by the operations you can perform on it. A number is something you can add and multiply; a string is something you can concatenate and take substrings of; a boolean is something you can negate, and so on. In a sense, users could already define their own types in early programming languages: you could create a record type date, for example, with integer fields for day, month and year. But what made abstract types new and different was the focus on operations: the user of the type would not need to worry about how its values were actually stored, in the same way that a programmer can ignore how the compiler actually stores integers. All that matters is the operations.

In Java, as in many modern programming languages, the separation between built-in types and user-defined types is a bit blurry. The classes in java.lang, such as Integer and Boolean are built-in; whether you regard all the collections of java.util as built-in is less clear (and not very important anyway). Java complicates the issue by having primitive types that are not objects. The set of these types, such as `int` and `boolean`, cannot be extended by the user.

These lecture notes have been collaboratively authored, with contributions from Saman Amarasinghe, Srini Devadas, Michael Ernst, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. *Copyright © MIT.*

Classifying Types and Operations

Types, whether built-in or user-defined, can be classified as mutable or immutable. The objects of a mutable type can be changed: that is, they provide operations which when executed cause the results of other operations on the same object to give different results. So `Date` is mutable, because you can call `setMonth` and observe the change with the `getMonth` operation. But `String` is immutable, because its operations create new string objects rather than changing existing ones. Sometimes a type will be provided in two forms, a mutable and an immutable form. `StringBuilder`, for example, is a mutable version of `String` (although the two are certainly not the same Java type, and are not interchangeable).

The operations of an abstract type are classified as follows:

- *Creators* create new objects of the type. A constructor may take an object as an argument, but not an object of the type being constructed.
- *Producers* create new objects from old objects of the type. The `concat` method of `String`, for example, is a producer: it takes two strings and produces a new one representing their concatenation.
- *Mutators* change objects. The `add` method of `List`, for example, mutates a list by adding an element to the end.
- *Observers* take objects of the abstract type and return objects of a different type. The `size` method of `List`, for example, returns an integer.

We can summarize these distinctions schematically like this:

creator: $t^* \rightarrow T$
producer: $T^+, t^* \rightarrow T$
mutator: $T^+, t^* \rightarrow void$
observer: $T^+, t^* \rightarrow t$

These show informally the shape of the signatures of operations in the various classes. Each T is the abstract type itself; each t is some other type. In general, when a type is shown on the left, it can occur more than once. For example, a producer may take two values of the abstract type; string concat takes two strings. The occurrences of t on the left may also be omitted; some observers take no non-abstract arguments (e.g., `size`), and some take several.

Here are some examples of abstract data types, along with their operations:

`int` is Java's primitive integer type. `int` is immutable, so it has no mutators.
creators: the numeric literals 0, 1, 2,
producers: arithmetic operators +, –, ×, ÷
observers: comparison operators ==, !=, <, >
mutators: none (it's immutable)

List is Java's list interface. List is mutable. List is also an *interface*, which means that other classes provide the actual implementation of the data type. These classes include ArrayList and Linked List.
creators: ArrayList constructor, LinkedList constructor, Collections.singletonList()
producers: Collections.unmodifiableList()
observers: size(), get()
mutators: add(), remove(), addAll(), Collections.sort()

String is Java's string interface. String is immutable.
creators: String(), String(char[]) constructors
producers: concat(), substring(), toUpperCase()
observers: length(), charAt()
mutators: none (it's immutable)

This classification gives some useful terminology, but it's not perfect. In complicated data types, there may be an operation that is both a producer and a mutator, for example. Some people use the term *producer* to imply that no mutation occurs.

A Zoo of Types

Here are some of the commonly-needed types that most programming language libraries provide. When you learn a new programming language, look for these in its library. The Java Collections API provides most of these. They're good tools for your toolbox.

type	overview	producers	observers	common reps
list	sequence for concatenation and front-append	add, append	first, rest, ith	array, linked list
queue	FIFO: first in, first out	enq, deq	first	array, list, circular buffer
stack	LIFO: last in, first out	push, pop	top	array, list
map	associates keys and values	put	get	association list, hash table, tree
set	unordered collection	insert, remove	contains	map, list, array, bitvector, tree
bag	like set, but element can appear more than once	insert, remove	count	map, array, association list

Designing an Abstract Type

Designing an abstract type involves choosing good operations and determining how they should behave. A few rules of thumb.

It's better to have a few, simple operations that can be combined in powerful ways than lots of complex operations.

Each operation should have a well-defined purpose, and should have a coherent behavior rather than a panoply of special cases. We probably shouldn't add a `sum` operation to `List`, for example. It might help clients who work with lists of `Integers`, but what about lists of `Strings`? Or nested lists? All these special cases would make `sum` a hard operation to understand and use.

The set of operations should be *adequate*; there must be enough to do the kinds of computations clients are likely to want to do. A good test is to check that every property of an object of the type can be extracted. For example, if there were no `get` operation, we would not be able to find out what the elements of a list are. Basic information should not be inordinately difficult to obtain. The

`size` method is not strictly necessary for List, because we could apply `get` on increasing indices until we get a failure, but this is inefficient and inconvenient.

The type may be generic: a list or a set, or a graph, for example. Or it may be domain-specific: a street map, an employee database, a phone book, etc. But it should not mix generic and domain-specific features. A `Deck` type intended to represent a sequence of playing cards shouldn't have a generic `add` method that accepts arbitrary objects (like integers or strings). Conversely, it wouldn't make sense to put a domain-specific method like `dealCards` into the generic type `List`.

Representation Independence

A good abstract data type should be *representation independent.* This means that the use of an abstract type is independent of its representation (the actual data structure or data fields used to implement it), so that changes in representation have no effect on code outside the abstract type itself. For example, the operations offered by `List` are independent of whether the list is represented as a linked list or as an array.

You won't be able to change the representation of an ADT at all unless its operations are fully specified with preconditions (requires), postconditions (effects), and frame conditions (modifies), so that clients know what to depend on, and you know what you can safely change.

Preserving Invariants

Finally, and perhaps most important, a good abstract data type should preserve its own invariants. An *invariant* is a property of a program that is always true. Immutability is one crucial invariant that we've already encountered: once created, an immutable object should always represent the same value, for its entire lifetime.

When an ADT preserves its own invariants, reasoning about the code becomes much easier. If you can count on the fact that `Strings` never change, you can rule out that possibility when you're debugging code that uses `Strings` — or when you're trying to establish an invariant for another ADT. Contrast that with a string class that guarantees that it will be immutable only if its clients promise not to change it. Then you'd have to check all the places in the code where the string might be used.

Immutability

We'll see many interesting invariants. Let's focus on immutability for now. Here's a specific example:

```
public class Transaction {
    public int amount;
    public Calendar date;

    public Transaction(int amount, Date date) {
        this.amount = amount;
        this.date = date;
    }
}
```

How do we guarantee that `Transaction` objects are immutable — that, once a transaction is created, its date and amount can never be changed?

The first threat to immutability comes from the fact that clients can (in fact, must!) directly access its fields. So nothing's stopping us from writing code like this:

```
Transaction t = new Transaction(10, new Calendar ());
```

```
t.amount += 10;
```

This is a trivial example of *representation exposure*, meaning that code outside the class can modify the representation directly. Rep exposure like this threatens not only invariants, but also representation independence. We can't change the implementation of `Transaction` without affecting all the clients who are directly accessing those fields.

Fortunately, Java gives us language mechanisms to deal with this kind of rep exposure:

```
public class Transaction {
    public int amount;
    public Calendar date;

    public Transaction(int amount, Calendar date) {
        this.amount = amount;
        this.date = date;
    }

    public int getAmount() {
        return amount;
    }

    public Calendar getDate() {
        return date;
    }
}
```

The private and public keywords indicate which fields and methods are accessible only within the class and which can be accessed from outside the class.

But that's not the end of the story: the rep is still exposed! Consider this (perfectly reasonable) client code that uses `Transaction`:

```
/** @return a transaction of same amount as t, one month later */
public static Transaction makeNextPayment(Transaction t) {
    Calendar d = t.getDate();
    d.add(Calendar.MONTH, 1);
    return new Transaction (t.getAmount(), d);
}
```

`makeNextPayment` takes a transaction and should return another transaction for the same amount but dated a month later. The `makeNextPayment` method might be part of a system that schedules recurring payments.

What's the problem here? The `getDate` call returns a reference to the *same* calendar object referenced by transaction `t`. So when the calendar object is mutated by add(), this affects the date in `t` as well:

<< snapshot diagram>>

`Transaction`'s immutability invariant has been broken. The problem is that `Transaction` leaked out a reference to a mutable object that its invariant depended on. We *exposed the rep*, in such a way that `Transaction` can no longer guarantee that its objects are immutable. Perfectly reasonable client code created a subtle bug.

We can patch this kind of rep exposure by *defensive copying*: making a copy of a mutable object to avoid leaking out references to the rep. Here's the code:

```
public Calendar getDate() {
    return (Calendar)date.clone();
}
```

clone() is probably the best way to do this with Calendar (despite the unfortunate problems with clone() in general – see Josh Bloch, *Effective Java*, item 10). Other classes offer a copy constructor, like StringBuilder(String).

But we're not done yet! There's still rep exposure. Consider this (again perfectly reasonable) client code:

```
/** @return a list of 12 monthly payments of identical amounts */
public static List<Transaction> makeYearOfPayments (int amount) {
    List<Transaction> list = new ArrayList<Transaction> ();
    Calendar date = new GregorianCalendar ();
    for (int i=0; i < 12; i++) {
        list.add (new Transaction (amount, date));
        date.add (Calendar.MONTH, 1);
    }
    return list;
}
```

This code intends to advance a single Calendar object through 12 months, creating a transaction for each date. But notice that the constructor of `Transaction` saves the reference that was passed in, so all 12 transaction objects end up pointing to the same date:

<< snapshot diagram >>

Again, the immutability of `Transaction` has been violated. We can fix this problem too by judicious defensive copying, this time in the constructor:

```
public Transaction(int amount, Calendar date) {
    this.amount = amount;
    this.date = (Calendar)date.clone();
}
```

In general, you should carefully inspect the argument types and return types of all your ADT operations. If any of the types are mutable, make sure your implementation doesn't return direct references to its representation.

You may object that this seems wasteful. Why make all these copies of dates? Why can't we just solve this problem by careful specification:

```
/**
 * ...
 * @param date Date of transaction.  Caller must never mutate date again!
 */
public Transaction(int amount, Calendar date) { ...
```

This approach is sometimes taken when there isn't any other reasonable alternative – for example, when the mutable object is too large to copy efficiently. But the cost in your ability to reason about the program, and your ability to *avoid bugs*, is enormous. In the absence of compelling arguments to the contrary, it's almost always worth it for an abstract data type to guarantee its own invariants, and preventing rep exposure is essential to that.

An even better solution is to prefer immutable types. If we had used an immutable date object instead of the mutable `Calendar`, then we would have ended this section after talking about `public` and `private`. No rep exposure would have been possible.

The Java Collections classes offer an interesting compromise: *immutable wrappers*. Collections.unmodifiableList() takes a (mutable) List and wraps it with an object that looks like a List, but whose mutators are disabled – set(), add(), remove() throw exceptions. So you can construct a list using mutators, then seal it up in an unmodifiable wrapper (and throw away your reference to the original mutable list), and get an immutable list. The downside here is that you get immutability at runtime, but not at compile time – Java won't warn you at compile time if you try to sort() this unmodifiable list. You'll just get an exception at runtime.

How to establish invariants

An invariant is a property that is true for the entire program – which in the case of an invariant about an object, reduces to the entire lifetime of the object.

If the object is a state machine, then we need to:

- establish invariant in the initial state
- ensure that all state transitions preserve the invariant

So your creators and producers must establish the invariant for new instances, and all mutators (and observers, too, but particularly mutators) must preserve it.

Immutable types are simpler, because they have only one state to reason about.

The risk of rep exposure makes the situation more complicated. So the full rule for proving invariants is:

Structural induction: If an invariant of an abstract data type is

(1) established by creators;

(2) preserved by producers, mutators, and observers;

and (3) no rep exposure occurs,

then the invariant is true of all instances of the abstract data type.

Rep Invariant and Abstraction Function

We now take a deeper look at the theory underlying abstract data types. This theory is not only elegant and interesting in its own right; it also has immediate practical application to the design and implementation of abstract types. If you understand the theory deeply, you'll be able to build better abstract types, and will be less likely to fall into subtle traps.

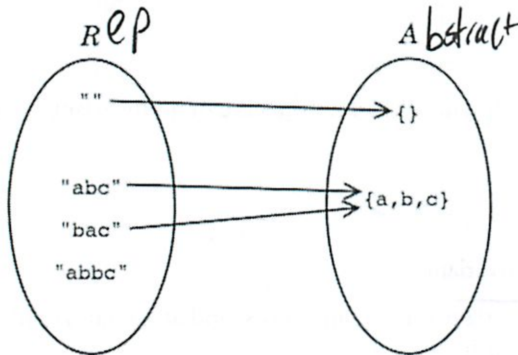
In thinking about an abstract type, it helps to consider the relationship between two spaces of values.

The space of *rep* or *representation* values consists of the values of the actual implementation entities. In simple cases, an abstract type will be implemented as a single object, but more commonly a small network of objects is needed, so this value is actually often something rather complicated. For now, though, it will suffice to view it simply as a mathematical value.

The space of *abstract* values consists of the values that the type is designed to support. These are a figment of our imagination. They're platonic entities that don't exist as described, but they are the way we want to view the elements of the abstract type, as clients of the type. For example, an abstract type for unbounded integers might have the mathematical integers as its abstract value space; the fact that it might be implemented as an array of primitive (bounded) integers, say, is not relevant to the user of the type.

Now of course the implementor of the abstract type must be interested in the representation values, since it is the implementor's job to achieve the illusion of the abstract value space using the rep value space.

Suppose, for example, that we choose to use a string to represent a set of characters. Then these form our two value spaces. We can show the two value spaces graphically, with an arc from a rep value to the abstract value it represents:



There are several things to note about this graph:

- Every abstract value is mapped to. The purpose of implementing the abstract type is to support operations on abstract values. Presumably, then, we will need to be able to create and manipulate all possible abstract values, and they must therefore be representable.
- Some abstract values are mapped to by more than one rep value. This happens because the representation isn't a tight encoding. There's more than one way to represent an unordered set of characters as a string.

Not all rep values are mapped. In this case, the string `"abbc"` is not mapped. If the type of threp is nontrivial, it will not make sense to give an interpretation for all rep values. A doubly-linked list representation, for example, can be twisted into all kinds of pretzel configurations that won't correspond to simple sequences, and for which we won't want to write special cases in the code. Or sometimes we will want to impose certain properties on the rep to make the code of the operations more efficient or easier to write. In this case, we have decided that the array should not contain duplicates. This will allow us to terminate the `remove` method when we hit the first instance of a particular character, since we know there can be at most one.

In practice, we can only illustrate a few elements of the two spaces and their relationships; the graph as a whole is infinite. So we describe it by giving two things:

An *abstraction function* that maps rep values to the abstract values they represent:

$$AF : R \rightarrow A$$

The arcs in the diagram show the abstraction function. In the terminology of functions, the properties we discussed above can be expressed by saying that the function is onto, not necessarily one-to-one, and often partial.

A *rep invariant* that maps rep values to boolean:

$$RI : R \rightarrow \text{boolean}$$

For a rep value *r*, *RI r* is true if and only if *r* is mapped by *AF*. In other words, *RI* tells us whether a given rep value is well-formed. Alternatively, you can think of *RI* as a set: it's the subset of rep values on which *AF* is defined.

A common confusion students have about abstraction functions and rep invariants is that they imagine that they are determined by the choice of rep and abstract value spaces, or even by the abstract value space alone. If this were the case, they would be of little use, since they would be saying something redundant that's already available elsewhere.

It's easy to see why the abstract value space alone doesn't determine *AF* or *RI*: there can be several representations for the same abstract type. A set of characters could equally be represented as a string, as above, or as a bit vector, with one bit for each possible character. Clearly we need two separate functions to map these two different rep value spaces.

It's less obvious why the choice of both spaces doesn't determine *AF* and *RI*. The key point is that defining a type for the rep, and thus choosing the values for the space of rep values, does not determine which of the rep values will be deemed to be legal, and of those that are legal, how they will be interpreted. Rather than deciding, as we did above, that the strings have no duplicates, we could instead allow duplicates, but at the same time require that the characters be sorted, appearing in nondecreasing order. This would allow us to perform a binary search on the string and thus check membership in logarithmic rather than linear time. Same rep value space — different rep invariant.

Even with the same type for the rep value space and the same rep invariant *RI*, we might still have different interpretations *AF*. Suppose *RI* admits any string of characters. Then we could define *AF*, as above, to interpret the array's elements as the elements of the set. But there's no *a priori* reason to let the rep decide the interpretation. Perhaps we'll interpret consecutive pairs of characters as subranges, so that the string `"acgg"` represents the set `{a,b,c,g}`.

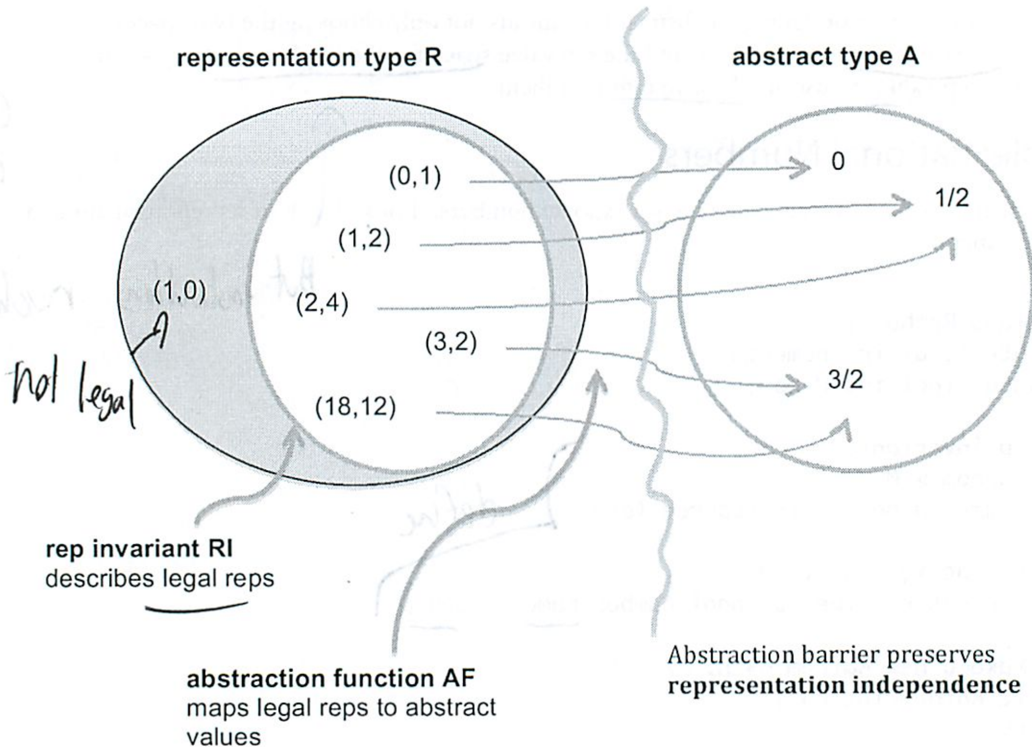
The essential point is that designing an abstract type means not only choosing the two spaces — the abstract value space for the specification and the rep value space for the implementation — but also deciding what rep values to use and how to interpret them.

Example: Rational Numbers

Here's an example of an abstract data type for rational numbers. Look closely at its rep invariant and abstraction function.

```
public class RatNum {
    private final int numer;
    private final int denom;

    // Rep invariant:
    //   denom > 0
    //   numer/denom is in reduced form

    // Abstraction Function:
    //   represents the rational number numer / denom

    /** Make a new Ratnum == n. */
    public RatNum(int n) {
        numer = n;
        denom = 1;
        checkRep();
    }

    /**
     * Make a new RatNum == (n / d).
     * @param n numerator
```

```
     * @param d denominator
     * @throws ArithmeticException if d == 0
     */
    public RatNum(int n, int d) throws ArithmeticException {
        // reduce ratio to lowest terms
        int g = gcd(n, d);
        n = n / g;
        d = d / g;

        // make denominator positive
        if (d < 0) {
            numer = -n;
            denom = -d;
        } else {
            numer = n;
            denom = d;
        }
        checkRep();
    }
}
```



Checking Rep Invariants and Implementing Abstraction Functions

The rep invariant isn't just a neat mathematical idea. If your implementation asserts the rep invariant at run time, then you can catch bugs early.

```
// Check that the rep invariant is true
// *** Warning: this does nothing unless you turn on assertion checking
// by passing -enableassertions to Java
private void checkRep() {
    assert denom > 0;
    assert gcd(numer, denom) == 1;
}
```

make sure always legal

will error at runtime (?) if not?

toString() is a useful place to implement the abstraction function:

```
/**
 * @return a string representation of this rational number
 */
// This effectively implements the abstraction function
public String toString() {
    return (denom > 1) ? (numer + "/" + denom) : (numer + "");
}
```

Summary

Abstract data types are characterized by their operations. Representation independence makes it possible to change the representation of a type without its clients being changed. An abstract data type that preserves its own invariants is easier and safer to use. Java language mechanisms like access control help ensure rep independence and invariants, but representation exposure is a trickier issue, and needs to be handled by careful programmer discipline.

6.005 Recitation

9/28

P-Set Questions

~~56 in (pt)~~ not valid

~~56 in pt~~

(56 in) pt ✓ is valid

(56) in = 56 in
(56) pt = 56 pt

~~2() +3~~

~~2 in 3pt +3~~

Should throw regular exceptions
- can put in sig

Can also do runtime exception
- no sig
- usually bad practice
- but can do it here

Lecture: Abstract Data Types

- Interface

Lecture 7
Recursive Data Types

10/3

- [ ] immutable lists
- [ ] datatypes + functions over types
- [ ] abstract syntax trees
- [ ] SAT

PS 4 out, due next Tuesday but hard

Abstract type of lists: ImList<E>

- immutable
- good for search problems like PS4
- Constructor ImList<E>
- empty void → ImList
  - empty() [ ]
- add ImList → ImList
  - add (0, empty()) [0]

↳ not an array, just a way of representing inside

ex: add(1, add(0, empty())) [1, 0]

↑ adds to front of list

- first ImList → E
- rest ImList → ImList

②

first(x)	1
rest(x)	[0]
rest(rest(x))	[]
first(rest(x))	0

Now how to actually implement:

Define arguments in interface
- can't put empty in since no static methods allowed
- can't define it in our interface instead
- So then in same file we actually have empty class
- - define all functions that can be done on it again
- Cons class
  - from Lisp from PDP-11
  - first and rest elements
    - are its rep (what it stores inside)

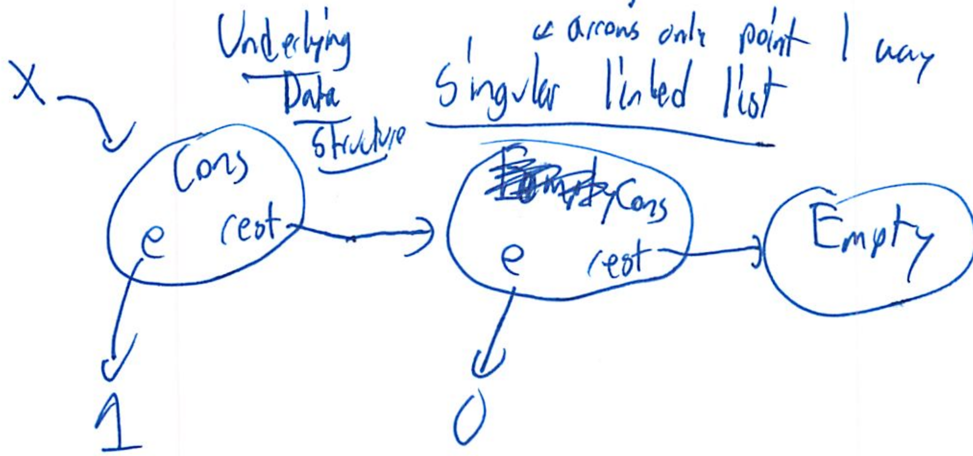
AM

(3)

Now rewrite our abstract list to Java

- don't want to call Cons
- part of rep
- want to hide Cons constructor
  - would want to make it private if possible

abstract	Java
empty()	new Empty<Integer>()
add(0, empty())	new Empty().add(0)
first(x)	x.first()



Singular good since it allows sharing

y = x.add

Will create new Cons element

(4) So X, Y share ^much of same data structure in memory

* Only works if Cons were immutable

We know this is happening since we are putting this in add

So its a recursive data type
- Cons can't make new Cons without already having one

Write some data type definitions

ImList = Empty + Cons(e : E, rest : ImList)

Grows from a seed of Empty list by Cons-ing onto it

Datatype definitions
- have datatype on left
- have variants on right separated by +
and denoted by constructor

For example

Cons(2, Cons(1, Cons(0, Empty)))

(5)

size : ImList → int

size(Empty) = 0

size(Cons(e : k, rest : ImList)) = 1 + size(rest)

↑define ↑recursive

rest is in rep of ~~cons~~ Cons
so we can use it

isEmpty : ImList → boolean

isEmpty(Empty) = true

isEmpty(Cons(...)) = false

append : ImList ~~*~~ ImList → ImList

[0] ...[1,2]... [0,1,2]

- 4 possible combos Empty, Cons
- append(Empty, list 2) = list 2 [ ] [...]
- append(Cons(e : E, rest : ImList), list2) = [...]

~~Cons(e, list2)~~ [e, rest] [...]

Cons(e, append(rest, list2))

6

Our size takes linear time

Can it do constant?

Store size in rep (cache) so effectively constant time

But does this violate "finalness"?

Immutable does not mean all fields are final

But also underlying objects must be non mutable

Not having mutator methods is good enough

So we can still change ourself inside by adding caching

- and still be immutable
- it's good mutability!

Can write rep invariant that size must be

0 (not calculated) or equal 1 + rest.size()

Know cache won't grow stale since can't change underlying variant

---

PS 4

- Solver for SAT
- Then solve sudoku

⑦

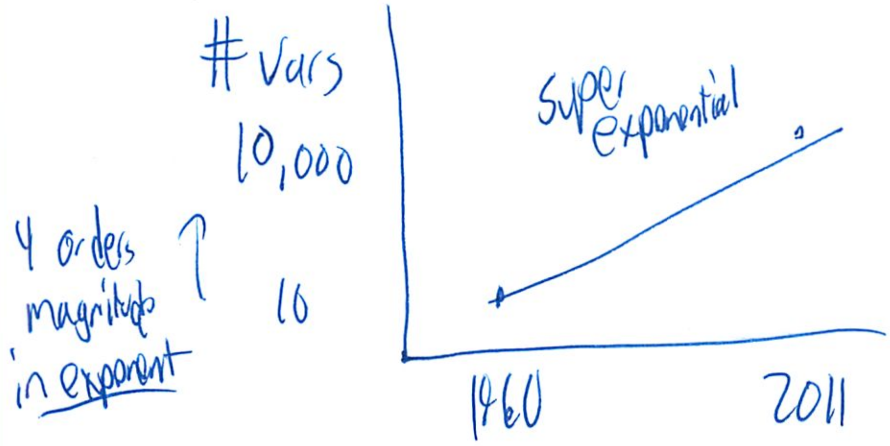
SAT = Satisfiability

- Simple concept
- Very hard to compute
- is there an assignment T/F that makes things true
- $(P \vee Q) \wedge (\neg P \vee R)$
- Try TT in a certain order

P	Q	R	
F	F	F	X
F	F	T	X
F	T	F	✓
	⋮		

- This is hard - exponential 2^k
- NP-complete problem
  - Very hard
- But in pratice eaiser to solve than theoritically

8



Only practical problems
not all theortical

So how to do in Java?

- more details in lecture notes + P-set instruction

Grammar

Formula = Var(name: String)
+ Not(f: Formula)
+ And (left: Formula, right: Formula)
+ Or

No paranthesis so abstract syntax tree

A more useful form CNF

AND of ORs

So $(P \vee Q) \wedge (\neg P \vee R)$

literal, literal — clause

Can define Formula in terms of clauses and literals

Formula = ImList <Clause>

Clause = ImList <Literals>

Literal = Positive (v: Var) + Negitive (v: Var)

↑ or

↑ use this data type in P-set

Read about backtrack SAT solver or recitation

6.005 Software Construction
Fall 2011
Prof. Rob Miller

L7: Recursive Data Types

Today

- Immutable lists
- Datatypes & functions over datatypes
- Abstract syntax trees
- SAT

Immutable Lists

Immutability is powerful not just because of its safety, but also because of the potential for sharing. Sharing actually produces performance benefits – less memory consumed, less time spent copying. Today we're going to look at how to represent list data structures a different way.

Let's define a data type for an immutable list, ImList<E>. The data type has four fundamental operations:

empty: void → ImList

// returns an empty list

add: E × ImList → ImList

// returns a new list formed by adding an element to the **front** of another list

first: ImList → E

// returns the first element of a list. requires the list to be nonempty.

rest: ImList → ImList

// returns the list of all elements of this list except for the first. requires the list to be nonempty.

These four operations have a long and distinguished pedigree. They are fundamental to the list-processing languages Lisp and Scheme (where for historical reasons they are called nil, cons, car, and cdr, respectively). They are widely used in functional programming, where you can often find them called head and tail instead of first and rest.

Before we design Java classes to implement this datatype, let's get used to the operations a bit, using lists of integers. For convenience, we'll write lists with square brackets, like [1,2,3], and we'll write the operations as if they are mathematical functions. Once we get to Java, the syntax will look different, but the operations will have the same meaning.

empty() = [ ]

add(0, empty()) = [0]

add(0, add(1, add(2, empty()))) = [0, 1, 2]

x ≡ add(0, add(1, add(2, empty()))) = [0, 1, 2]

first(x) = 0

rest(x) = [1, 2]

These lecture notes have been collaboratively authored, with contributions from Saman Amarasinghe, Srini Devadas, Michael Ernst, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. Copyright © MIT.

first(rest(x)) = 1

rest(rest(x)) = [2]

first(rest(rest(x)) = 2

rest(rest(rest(x))) = [ ]

Fundamental relationship between first, rest, and add:

first(add(e, l)) = e

rest(add(e, l)) = l

What *add* puts together, *first* and *rest* peel back apart.

Immutable Lists in Java

To implement this datatype, we'll use an interface:

```
public interface ImList<E> {
    public ImList<E> add(E e);
    public E first();
    public ImList<E> rest();
}
```

and two classes that implement it. Empty represents the result of the empty operation (an empty list), and Cons represents the result of an add operation (an element glued together with another list):

```
public class Empty<E> implements ImList<E> {
    public Empty() { }
    public ImList<E> add(E e) { return new Cons<E>(e, this); }
    public E first() { throw new UnsupportedOperationException(); }
    public ImList<E> rest() { throw new UnsupportedOperationException(); }
}

public class Cons<E> implements ImList<E> {
    private E e;
    private ImList<E> rest;

     public Cons(E e, ImList<E> rest) {
        this.e = e;
        this.rest = rest;
    }
    public ImList<E> add(E e) {
        return new Cons<E> (e, this);
    }
    public E first() {
        return e;
    }
    public ImList<E> rest() {
        return rest;
    }
}
```

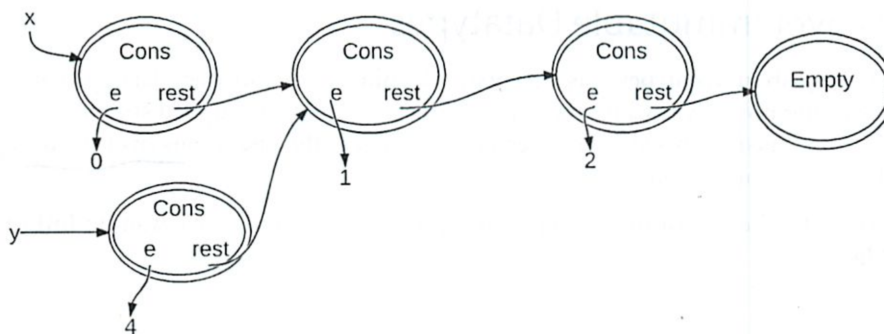
So we've got methods for *add*, *first*, and *rest*, but where is the fourth operation of our datatype, *empty*? Unfortunately Java makes it hard. The right way to do it is as a static method that takes no arguments and produces an instance of Empty. We'd like to add this to the ImList interface along with the other operations, but Java doesn't allow any method bodies in an interface, not even static

methods. So we'll sacrifice some representation independence (more on this later) and use the Empty class constructor directly to represent the *empty* operation. So to do *empty*, you write new Empty().

Here's some actual Java code that parallels the abstract examples we wrote earlier:

Java syntax	Functional syntax	Result
`ImList<Integer> nil =` `    new Empty<Integer>();`	`nil ≡ empty()`	`[ ]`
`nil.add(0)`	`add(0, nil)`	`[0]`
`nil.add(0).add(1).add(2)`	`add(0, add(1, add(2, nil)))`	`[0, 1, 2]`
`ImList<Integer> x =` `   nil.add(0).add(1).add(2);`	`x ≡ add(0, add(1, add(2, nil)))`	`[0, 1, 2]`
`x.first()`	`first(x)`	`0`
`x.rest()`	`rest(x)`	`[1, 2]`
`x.rest().first()`	`first(rest(x))`	`1`
`x.rest().rest()`	`rest(rest(x))`	`[2]`
`x.rest().rest().first()`	`first(rest(rest(x))`	`2`
`x.rest().rest().rest()`	`rest(rest(rest(x)))`	`[ ]`
`ImList<Integer> y =` `  x.rest().add(4);`	`y ≡ add(4, rest(x))`	`[4, 1, 2]`

Let's look at a snapshot diagram of what x and y would look like:



The key thing to note here is the sharing of structure that the immutable list provides.

Two Classes, One Interface

Note that this design is different from what we saw with List, ArrayList, and LinkedList. List is an abstract data type and ArrayList and LinkedList are two *alternative* concrete representations for that datatype.

For ImList, the two implementations Empty and Cons *cooperate* in order to implement the datatype – you need them both.

Recursive Datatype Definitions

The abstract data type ImList, and its two concrete classes Empty and Cons, form a *recursive* data type. Cons is an implementation of ImList, but it also uses ImList inside its own rep (for the rest field), so it recursively requires an implementation of ImList in order to successfully implement its contract.

To make this fact clearly visible, we'll write a **datatype definition**:

ImList = Empty + Cons(first:E , rest:ImList)

This is a recursive definition of ImList set of values. Read it like this: the set ImList consists of values formed in two ways: either by the Empty constructor, or by applying the Cons constructor to an element and an ImList. The recursive nature of the datatype becomes far more visible when written this way.

We can also write ImList values as *terms* or *expressions* using this definition, e.g.:

Cons(0, Cons(1, Cons(2, Empty)))

Formally, a datatype definition has:

- the datatype on the left
- **variants** of the datatype separated by + on the right
- each variant is a constructor with zero or more named (and typed) arguments

Another example is a binary tree:

Tree = Empty + Node(e:E, left:Tree, right:Tree)

We'll see more examples later.

Functions over Immutable Datatypes

This way of thinking about datatypes – as a recursive definition of an abstract datatype with concrete variants – is appealing not only because it can handle recursive and unbounded structures like lists and trees, but also because it provides a convenient way to describe operations over the datatype, as functions with one case per variant.

For example, consider the size of the list, which is certainly an operation we'll want in ImList. We can define it like this:

size : ImList → int

// returns the size of the list

and then fully specify its meaning by defining it for each variant of ImList:

size(Empty) = 0

size(Cons(first: E, rest: ImList)) = 1 + size(rest)

We can think about the execution of size on a particular list as a series of reduction steps:

size(Cons (0, Cons (1, Empty)))

= 1 + size(Cons (1, Empty))

= 1 + (1 + size(Empty))

= 1 + (1 + 0)

= 2

```
    private final ImList<E> rest;
    private int size = 0;
    // rep invariant:
    //   rest != null
    //   size > 0 implies size == 1+rest.size()
    ...
    public int size() {
        if (size == 0) size = 1 + rest.size();
        return size;
    }
}
```

Note that we're using the special value 0 (which can never be the size of a Cons) to indicate that we haven't computed the size yet. Note also that this change introduces a new clause to the rep invariant, relating the size field to the rest field.

There's something interesting happening here: this is an immutable datatype, and yet it has a mutable rep. It's modifying its own size field, in this case to cache the result of an expensive operation. This is an example of a **beneficient mutation**, a state change that doesn't change the abstract value represented by the object, so the type is still immutable.

Rep Independence and Rep Exposure Revisited

Does ImList still have rep independence? Yes and no. We really want to hide the classes Empty and Cons from the client. For simplicity, we exposed the constructor of Empty. We could still hide Cons, though, by making it package-private, so that classes outside ImList's package can't see it or use it.

But we do still have a lot of freedom. The internal rep of Cons could add a size field. We could even have an extra array in there to make get() run fast! This would get pretty expensive in space, though.

Is there rep exposure because Cons.rest() returns a reference to its internal list? Could a clumsy client add elements to the rest of the list? If so, this would threaten two of Cons's invariants: that it's immutable, and that the cached size is always correct. But there's no risk of rep exposure, because the internal list is immutable. Nobody can threaten the rep invariant of Cons.

Null vs. Empty

It might be tempting to get rid of the Empty class and just use null instead. Resist that temptation.

Using an object, rather than a null reference, to signal the base case or endpoint of a data structure is an example of a design pattern called *sentinel objects.* The enormous advantage that a sentinel object provides is that it acts like an object in the datatype, so you can call methods on it. So we can call the size() method even on an empty list. If empty lists were represented by null, then we wouldn't be able to do that, and as a consequence our code would be full of tests like:

```
if (lst != null) n = lst.size();
```

which clutter the code, obscure its meaning, and are easy to forget. Better the much simpler

```
n = lst.size();
```

which will always work if an empty lst refers to an Empty object.

Keep nulls out of your data structures, and your life will be happier.

And these cases can be translated directly into Java as methods in ImList, Empty, and Cons:

```
public interface ImList<E> {
    ...
    public int size();
}

public class Empty<E> implements ImList<E> {
    ...
    public int size() { return 0; }
}

public class Cons<E> implements ImList<E> {
    ...
    public int size() { return 1 + rest.size(); }
}
```

Let's try a few more examples:

isEmpty : ImList → boolean

> isEmpty(Empty) = true
>
> isEmpty(Cons(first: E, rest: ImList)) = false

contains : ImList x E → boolean

> contains(Empty, e: E) = false
>
> contains(Cons(first: E, rest: ImList), e: E) = (first = e) ∨ contains(rest, e)

get: ImList x int → E

> get(Empty, e: E) = *undefined*
>
> get(Cons(first: E, rest: ImList), n) = if n=0 then first else get(rest, n-1)

append: ImList x ImList → ImList

> append(Empty, list2: ImList) = list2
>
> append(Cons(first: E, rest: ImList), list2: ImList) = add(first, append(rest, list2))

reverse: ImList → ImList

> reverse(Empty) = empty()
>
> reverse(Cons(first: E, rest: ImList)) = append(rest, reverse(add(first, empty()))

For reverse, it turns out that the recursive definition produces a pretty bad implementation in Java, with performance that's quadratic in the length of the list you're reversing. We can rewrite that better using an iterative approach.

Tuning the Rep

Getting the size of a list is a common operation. Right now our implementation of size() takes O(n) time, where n is the length of the list. We can make it better with a simple change to the rep of the list that caches the size the first time we compute it, so that subsequently it costs only O(1) time to get:

```
public class Cons<E> implements ImList<E> {
    private final E e;
```

Declared Type vs. Actual Type

Now that we're using interfaces and classes, it's worth a moment to reinforce an important point about how Java's type-checking works. In fact every statically-checked object-oriented language works this way.

There are two worlds in type checking: **compile time** before the program runs, and **run time** when the program is executing.

At compile time, every variable has a **declared type**, stated in its declaration. The compiler uses the declared types of variables (and method return values) to deduce declared types for every expression in the program.

At run time, every object has an **actual type**, imbued in it by the constructor that created the object. For example, new String() makes an object whose actual type is String. new Empty() makes an object whose actual type is Empty. new ImList() is forbidden by Java, because ImList is an interface – it has no object values of its own, and no constructors.

Datatype Definitions vs. Grammars

You can think of datatype definitions are being like little grammars -- just like the kind you've seen for specifying the form of programs, or commands to a shell, etc. But unlike grammars, which show the concrete form the objects take (ie, their physical appearance – e.g. including parentheses and curly braces), datatype definitions represent only the conceptual shape. Put another way, datatype definitions don't help you parse, but are good for representing the results of parsing.

A parser for a textual language usually produces a tree-shaped datatype. This datatype is called an *abstract syntax tree*, or AST. It's called abstract to distinguish it from a concrete syntax tree that directly follows the syntax. An abstract syntax tree may omit details (like extra pairs of parentheses or curly braces) or expand syntactic sugar.

The datatype definition for an AST, despite being abstract, can take its shape directly from the grammar. Recall our grammar for HTML markup:

Html ::= (Normal | Italic) *

Italic ::= <i> Html </i>

Normal ::= Text

Text ::= [^ _]*

Let's use this grammar to define a recursive datatype for Html. Here's one possibility, which uses an ImList to represent the repetition in the Html production, and then needs a new datatype to represent Normal | Italic expression:

Html = ImList<Node>

Node = Normal(content: String) + Italic(content: Html)

Here's the code:

```
public class Markup {
    public class Html {
        private final ImList<Node> nodes;
        public Html(ImList<Node> nodes) { this.nodes = nodes; }
    }
```

```
    public interface Node {
    }

    public class Normal {
        private final String content;
        public Normal(String content) { this.content = content; }
    }

    public class Italic {
        private final Html content;
        public Italic(Html content) { this.content = content; }
    }
}
```

Now let's implement some functions over this datatype.

length : Html → int

> // returns number of characters of text
>
> length(Cons(first: Node, rest: ImList) = true
>
> isEmpty(Cons(first: E, rest: ImList)) = false

scramble : Html → String

> // returns text in the tree concatenated together,
> // with each italic region randomly scrambled

Boolean Formulas and Satisfiability

the SAT problem

- given a formula made of boolean variables and operators $(P \lor Q) \land (\neg P \lor R)$
- find an assignment to the variables that makes it true
- possible assignments, with solutions in green, are:

> {P = false, Q = false, R = false}
>
> {P = false, Q = false, R = true}
>
> {P = false, Q = true, R = false}
>
> {P = false, Q = true, R = true}
>
> {P = true, Q = false, R = false}
>
> {P = true, Q = false, R = true}
>
> {P = true, Q = true, R = false}
>
> {P = true, Q = true, R = true}

Truth table

SAT solver

- program that takes a boolean formula in CNF
- returns an assignment, or says none exists

how to build a SAT solver, version one

- just enumerate assignments, and check formula for each
- for k variables, 2^k assignments: surely can do better?

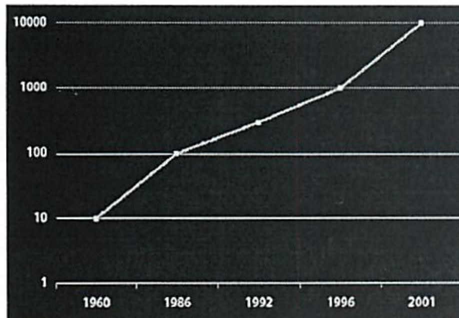
SAT is hard

- in the worst case, no: you can't do better
- Cook (1973): 3-SAT (3 literals/clause) is "NP-complete"
- the quintessential "hard problem" ever since

how to be a pessimist

- suppose you have a problem P (that is, a class of problems)
- show SAT reducible to P (ie, can translate any SAT-problem to a P-problem)
- then if P weren't hard, SAT wouldn't be either; so P is hard too

remarkable discovery



#boolean vars SAT solver can handle (from Sharad Malik)

- most SAT problems are easy
- can solve in much less than exponential time

how to be an optimist

- suppose you have a problem P
- reduce it to SAT, and solve with SAT solver

In the 1980s, researchers were publishing papers on how to find hard SAT problems! It turned out that even though in the worst case SAT is really hard, in practice almost all the cases you get if you generate them randomly are easy. And the ones that arise in real problems are often easy too. The story's actually more complicated than this though; it turns out that there's what's called a 'phase transition', a point at which problems get really hard, and this phase transition is roughly at the midpoint between the two extremes of the formula being so constrained it's easy to solve because you can easily determine values for variables early on, and the formula being so underconstrained that it's easy to solve just by guessing.

applications of SAT

planning

- solve (initial state ∧ goal ∧ rules) to obtain plan
- eg, ZYpp package manager for Linux

verification

- solve (code ∧ ¬ spec) to obtain counterexample
- industrial application to hardware; software applications coming
- eg, Cadence Incisive hardware verifier

design

- solve (design rules ∧ constraints ∧ requirements) to obtain design

for more info

- see http://www.satlive.org

why are we teaching you this?

SAT is cool

- good for (geeky) cocktail parties
- many useful applications
- compilation-to-SAT idea is powerful
- builds on your 6.042 knowledge

fundamental techniques

- you'll learn about datatypes and functions
- same ideas will work for any compiler or interpreter

A Naive SAT Solver

one way to represent boolean formulas

Formula = Var(name:String)
+ Not(formula: Formula)
+ Or(left: Formula,right: Formula)
+ And(left: Formula,right: Formula)

(P∨Q)∧(¬P∨R) would be

```
And( Or(Var("P"), Var("Q")),
     Or(Not(Var("P")), Var("R")) )
```

Socrates⇒Human ∧ Human⇒Mortal ∧ ¬ (Socrates⇒Mortal) would be:

```
And( Or(Not(Var("Socrates")), Var("Human")),
     And ( Or(Not(Var("Human")), Var("Mortal")),
           Not( Or(Not(Var("Socrates")),Var("Mortal")))))
```

Note that a client of this datatype should NOT see the internal classes, Not, Or, and And. They should use abstract operations to build the formula. In this case they would be operators like *and*, *or*, and *not*. We have to make an exception with the Var class, and expose it, or else use a constructor method. So here's what a client might write in Java:

$P \vee Q$ new Var("P") .or (new Var("Q"))

$\neg P \vee R$ (new Var("P").not()) .or (new Var("R"))

a naive SAT solver

generate and test strategy

- steps
  1. extract set of variables from formula
  2. try all assignments of true/false values to those vars
     a. we'll represent an assignment with an *environment* Env, which is just a list of variables and their values
  3. evaluate the formula for each environment
  4. return the first environment in which the formula evaluates to true

G + √

- functions we'll need

  vars: Formula → Set<Var>

  solve: Formula → Env?

  eval: Formula, Env → Boolean

new datatypes we'll need

Set<T> = ImList<T>

Env = ImList<Var × Boolean>

Boolean = True + False + Undefined

what's wrong with our solver?

consider formula

$Socrates \Rightarrow Human \wedge Human \Rightarrow Mortal \wedge \neg (Socrates \Rightarrow Mortal)$

suppose order of trying the variables is Socrates, Human, Mortal

and suppose we set Socrates to true

then clearly must set Human to true

and then must set Mortal to true...

...but our solver ignores all this!

A better SAT Solver

Conjunctive Normal Form (CNF)

conjunctive normal form (CNF) or "product of sums"

(P∨Q)∧(¬P∨R) is in conjunctive normal form.

- set of clauses, each containing a set of literals {{P, Q}, {¬P, R}}
- literal is just a variable, maybe negated

Note that CNF is just a format for a boolean formula -- but one that turns out to be very helpful, making it easier to write solvers. The notion of a literal is important, since it means you can only negate variables, and not clauses.

Datatype definition:

    Formula = ImList<Clause>   // a list of clauses ANDed together
    Clause = ImList<Literal>   // a list of literals ORed together
    Literal = Positive(v: Var) + Negative(v:Var)   // either a variable P or its negation ¬P
    Var = String

Note that as long as the concrete classes we were using in the old Formula (And, Or, Not) were hidden from the client, and the client was limited to using abstract operations to combine formulas (and, or, not), then we can freely make this change to the rep without changing any client code. e .g.,

    ¬P∨R   (new Var("P").not()) .or (new Var("R"))

now constructs a data structure that looks like:

    [[Negative(Var("P")), Positive(Var("R"))]]

That is, a list containing a single clause, which in turn contains two literals, one negative and one positive.

basic backtracking algorithm using CNF

- CNF is a product of sums: we need every clause true, and at least one literal in each clause

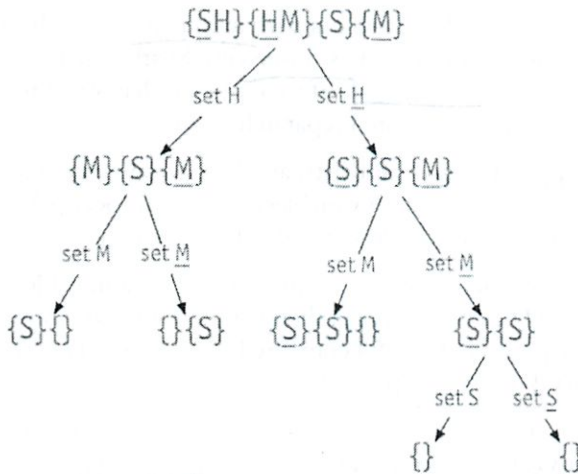
elements of the algorithm

- backtracking search: pick a literal, try false then true
- if clause set is empty, success
- if clause set contains empty clause, failure

example

- want to prove Socrates⇒Mortal from Socrates⇒Human ∧ Human⇒Mortal
- so give solver: Socrates⇒Human ∧ Human⇒Mortal ∧¬ (Socrates⇒Mortal)
- in CNF: (¬Socrates ∨ Human) ∧ (¬Human ∨ Mortal) ∧ Socrates ∧ ¬Mortal
- in clausal form: {{¬Socrates,Human},{¬Human,Mortal},{Socrates},{¬Mortal}}
- in shorthand: {$\underline{S}$H} {$\underline{H}$M} {S} {$\underline{M}$} (underlines mean negation)

backtracking execution



- stop when node contains {} (failure) or is empty (success)
- in this case, all paths fail, so theorem is valid
- in worst case, number of leaves is 2^#literals

DPLL: the classic SAT algorithm

- Davis-Putnam-Logemann-Loveland, 1962

key idea: **unit propagation** on top of backtracking search

- if a clause contains one literal, set that literal to true

example

{SH}{HM}{S}{M}
unit S
{H}{HM}{M}
unit H
{M}{M}
unit M
{}

did in recitation; its like a shortcut for when 1 litteral

- in this case, no splitting needed
- propagate S, then H, then M
- performance is often much better, but worst case still exponential

Backtracking Search with Immutability

A final comment about what we've seen in this lecture. We started out with immutable lists, which are a representation that permits a lot of sharing between different list instances. Sharing of a particular kind, though: only the ends of lists can actually be shared. If two lists are identical at the beginning but then diverge from each other, they have to be stored separately. (Why?)

It turns out that backtracking search is a great application for these lists, and here's why. A search through a space (like the space of assignments to a set of boolean variables) generally proceeds by making one choice after another, and when a choice leads to a deadend, you backtrack.

Mutable data structures are typically not a good approach for backtracking. If you use a mutable Map, say, to keep track of the current variable bindings you're trying, then you have to undo those bindings every time you backtrack. That's error-prone and painful compared to what you do with immutable maps – when you backtrack, you just throw the map away!

But immutable data structures with no sharing aren't a great idea either, because the space you need to keep track of where you are (in the case of SAT, the environment) will grow quadratically if you have to make a complete copy every time you take a new step. You need to hold on to all the previous environments on your path, in case you need to back up.

Immutable lists have the nice property that each step taken on the path can share all the information from the previous steps, just by adding to the front of the list. When you have to backtrack, you stop using the current step's state – but you still have references to the previous step's state.

Perhaps best of all, a search that uses immutable data structures is immediately ready to be parallelized. You can delegate multiple processors to search multiple paths at once, without having to deal with the problem that they'll step on each other in a shared mutable data structure. We'll talk about this more when we get to concurrency.

hmm - would have to think about more

Summary

big ideas

- datatype definitions: a powerful way to think about abstract data types, particularly recursive ones
- backtracking search: easy with immutable types
- SAT: an important problem, theoretically & practically

```java
package inclass;
@SuppressWarnings("unused")
public class Formula1 {
    // datatype definition:
    //    Formula = Var(name:String)
    //              + Not(f:Formula)
    //              + And(left:Formula, right:Formula)
    //              + Or(left:Formula, right:Formula)

    public static interface Formula {
//      operations:
//       public Formula and(Formula that);
//       public Formula or(Formula that);
//       public Formula not();
//       etc.
    }

    public static class Var implements Formula{
        private final String name;
        public Var(String name) { this.name = name; }
    }

    public static class And implements Formula{
        private final Formula left, right;
        public And(Formula left, Formula right) { this.left = left; this.right = right; }
    }

    public static class Or implements Formula{
        private final Formula left, right;
        public Or(Formula left, Formula right) { this.left = left; this.right = right; }
    }

    public static class Not implements Formula{
        private final Formula f;
        public Not(Formula f) { this.f = f; }
    }


}
```

```java
package inclass;
import static beforeclass.Im.*;
@SuppressWarnings("unused")
public class Formula2 {
    // datatype definitions:
    //    Formula = ImList<Clause>
    //    Clause = ImList<Literal>
    //    Literal = Positive(v:Var) + Negative(v:Var)
    //    Var = String

    public static class Formula {
        private final ImList<Clause> clauses;
        public Formula() { clauses = new Empty<Clause>(); }
//      operations:
//      public Formula and(Formula that);
//      public Formula or(Formula that);
//      public Formula not();
//      etc.
    }

    public static class Clause {
        private final ImList<Literal> literals;
        public Clause() { literals = new Empty<Literal>(); }
    }

    public static interface Literal {
    }
    public static class Positive implements Literal {
        private final Var var;
        public Positive(Var var) { this.var = var; }
    }
    public static class Negative implements Literal {
        private final Var var;
        public Negative(Var var) { this.var = var; }
    }

    public static class Var {
        private final String name;
        public Var(String name) { this.name = name; }
    }

}
```

```java
package inclass;

// ImList, Empty, and Cons are nested inside one class
// for ease of presentation in lecture.  A real program would
// put them in separate files. Also, specs omitted to save space --
// see the lecture notes for specs.
public class Im {
    // datatype definition:
    //    ImList = Empty + Cons(e:E, rest:ImList)

    public static interface ImList<E> {
        public ImList<E> add(E e);
        public E first();
        public ImList<E> rest();
        public int size();
    }

    public static class Empty<E> implements ImList<E> {
        public Empty() { }
        public ImList<E> add(E e) { return new Cons<E>(e, this); }
        public E first() { throw new UnsupportedOperationException(); }
        public ImList<E> rest() { throw new UnsupportedOperationException(); }
        public int size() { return 0; }
    }

    public static class Cons<E> implements ImList<E> {
        private final E e;
        private final ImList<E> rest;
        private int size = 0; // caches the size of list
        // rep invariant:
        // size ==0 (in which case we don't know the size yet)
        //   || size == 1+rest.size()
        public Cons(E e, ImList<E> rest) { this.e = e; this.rest = rest; }
        public ImList<E> add(E e) { return new Cons<E> (e, this); }
        public E first() { return e; }
        public ImList<E> rest() { return rest; }
        public int size() {
            if (size == 0) size = 1 + rest.size();
            return size;
        }
    }

}
```

```java
package inclass;
import static beforeclass.Im.*;
@SuppressWarnings("unused")
public class Markup {
    // datatype definitions:
    //     Html = ImList<Node>
    //     Node = Normal(content:String) + Italic(content:String)

    public static class Html {
        private final ImList<Node> nodes;
        public Html(ImList<Node> nodes) { this.nodes = nodes; }
    }

    public static interface Node {
    }

    public static class Normal {
        private final String content;
        public Normal(String content) { this.content = content; }
    }

    public static class Italic {
        private final Html content;
        public Italic(Html content) { this.content = content; }
    }

}
```

6.005 Recitation 7

SAT - satisfibility
- Can that Boolean be made True?
- Could brute force every permutation
  - but 2^n
- Or put to CNF then test every statement

example

Socraties → Human ∧

Human → Mortal ∧

¬(Socraties → Mortal)

We look at this and guess never SAT
But how to do this mathmatically

First part Socaties → Human
(¬S ∨ H) ← is mathmatically

②

So all together

$$(\neg S \vee H) \overset{\downarrow AND}{\wedge} (\neg H \overset{\downarrow OR}{\vee} M) \wedge \neg(\neg S \vee M)$$

This is not CNF since last part is NOT CNF

So use good ol' Mr. DeMorgan

Or we could just remove ()

$$(\neg S \vee H) \wedge (\neg H \vee M) \wedge (S) \vee (\neg M)$$

We could check everything in TT

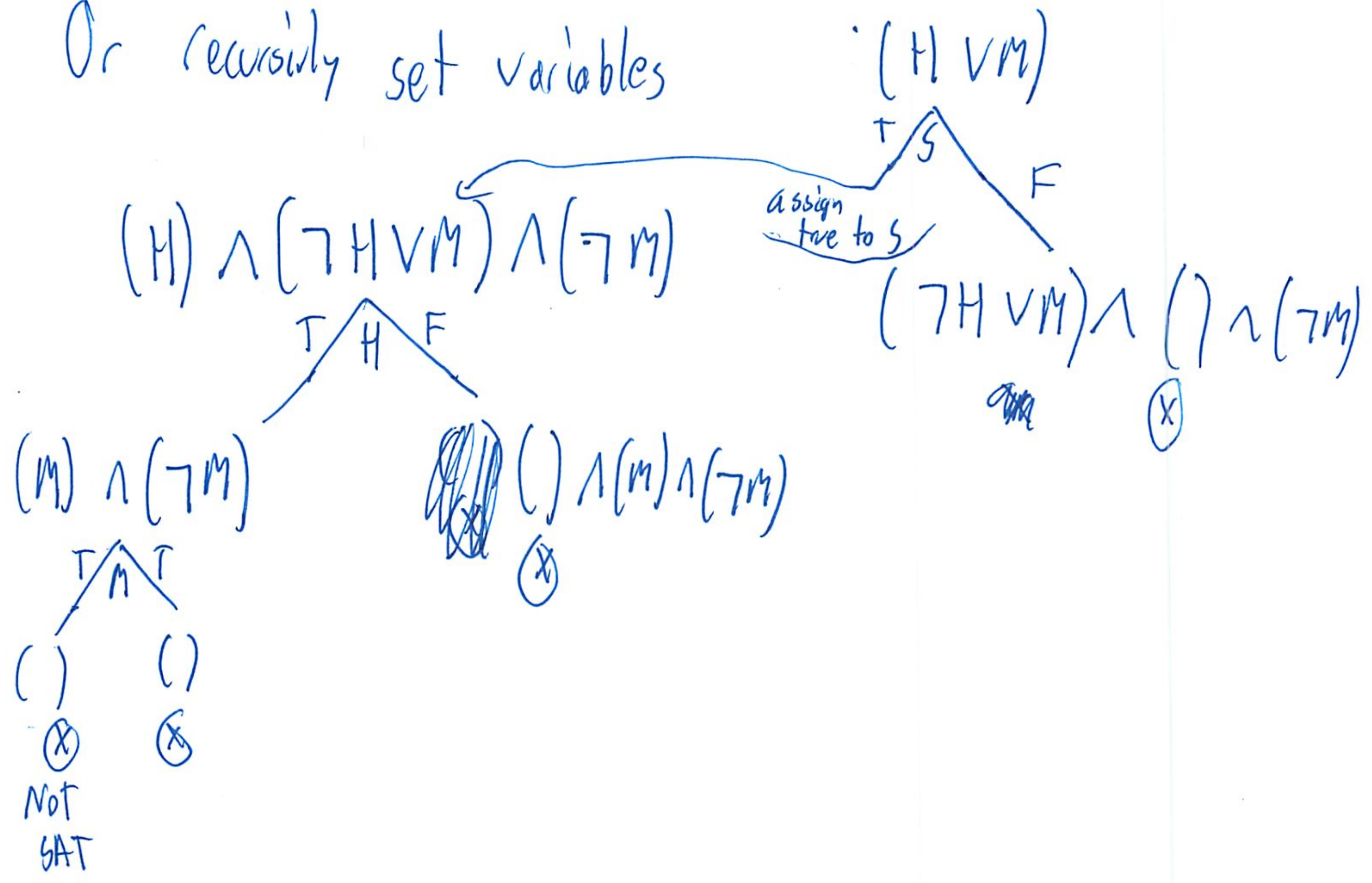
S	H	M	
T	T	T	F
⋮			⋮

Or look for shortcuts. Clauses w/ 1 variables

— so S always T, M never true

③

Or recursivly set variables



This is better - not doing <u>every</u> possibility
But still kinda inefficent

Now go back to single units shortcut
- clauses w/ a single literal

Need to SAT each clause
Only way to SAT ~~each~~ clause is to make each clause true

④

So $S = T$

Now what do we have? — Simplify

$(H) \wedge (\neg H \vee M) \wedge (\neg M)$

Now can set $H = T$

Simplify

$(M) \wedge (\neg M)$

⊗ does not work

Could also have started w/ $M = F$

$(\neg S \vee H) \wedge (\neg H) \wedge (S)$

etc.

Same result

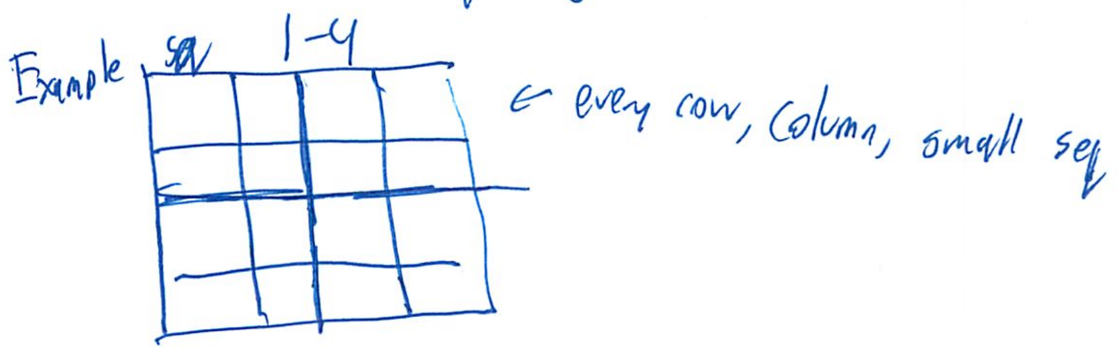
Could then do it with shorter clauses if no 1-clause

And make tree

Like before

(5)

PS 3 is with squares



How translate this into boolean formula?

- want to uniquely describe a grid

Want to represent it uniquely

$(S1 \otimes S2) \otimes (S3 \otimes S4)$

Or use 2 Booleans for each square

Must use 64 booleans for whole thing?

(Did not give answer)

Did not like my AND everything

$R_1 = A_{R_1} \cap B_{R_1} \cap C_{R_1} \cap D_{R_1}$

⋮ for each row and column

(6) And $R1 \cap R2 \cap R3 \ldots \cap C1 \cap C2 \ldots \cap B1 \cap B2 \ldots$

Said needs to uniqily represent square

Collabcode 7

Expr ::= Compand | Value

Compand ::= Expr Op Expr

recursive data type

Value ::= double

Op ::= + | - | x / ÷

Store internal double and just return it

6.005

Lecture 8

Interperters + Visitors

10/5

- ☐ design patterns
- ☐ interpreter patterns
- ☐ visitors patter

PS 4 due Tue @ Midnight ~ LA hrs Tue

no lecture Mon or Wed

Review session here Wed in class

Q1 Friday Walker 11AM

Using alpha.classmetric.com trial today

Today is last lecture before quiz

Don't copy each other solution byte for byte

No Java on Piazza

Design Patterns - tools in toolbox for programming

Last time looked for Formulas $(P \vee Q) \wedge (\neg P \vee R)$ for PS 4

(2)

Today's data types not for PSY

Abstract syntax tree

Formula = Var (name : String)
+ Not (f : Formula)
+ AND (left, right : Formula)
+ OR

Have interface for whole thing
Have class for each part

eval : Formula x Env → boolean

eval (Var (name : String), env) = env. lookup (name)

eval (Not (f : Formula), env) = ¬ env (f, env)

eval (And / Or (left, right : Formula), env) = eval (left, env) ∧ AND / ∨ OR eval (right, env)

③

Call this function as method
  ?implemented on all invariants

Called <u>Interpreter Design Pattern</u>
  - commonly seen on interpreters for code

But code is divided up into bunch of files

Makes it much harder to understand
  - often in seperate files

Like AND, OR are very similar
  - if change one, would want to change other one

Now want to bring all together

And want fn hasNot : Formula → boolean

  How do you do that in interface?

  You will keep doing that over + over again. - making fns

  hasNot(Var(name: str)) = false

  hasNot(Not(f)) - true

  hasNot(And/Or(l, r: f)) = hasNot(l) ∨ hasNot(r)

(4)

Don't want to add special support for each function

~~What~~ Want to say (Java does not support)

```
boolean hasNot(Formula f) {
    switch (f) {
        case Var v: return false;
        case Not n: return true
        case And a: return hasNot(a.left) || hasNot(a.right)
             Or o:                        ||
```

1 all 1 place
not multiple
files

Could use instanceOf - but kinda bad

Declared types vs actual types

- for Formula can only do what is defined in interface
- ~~Son~~ but we checked its an AND
- need to tell Java that

    And a = (And) f;

- Called a down cast

(5)

But is bad style - not safe
Not statically checked

New varients cald appear
- ForAll ()
- thereExists ()

~~(A)~~ w/ Interper Pattern putting items in interface must fix since gives error

Otherwise must remeber to add yourself manually
- not safe from bugs, read for change

Its good to have a default that finds error
So can find error at runtime

So we can use Visitor Pattern

- interfaces
  - use parameter
- has Not classs
- then write functions for each class individually.

Interperter/Java handels switch

(6)

Using accept in interface

public <R> R accept (Visitor<R> v)

↑ must put type param for method

then Need to implement accept in each class

```
public <R> R accept(Visitor <R> v) { return v.onVar(this); }
```

Now HasNot is of type Var, not formula
that did the downcast

Just passing along to node itself

Now how to run it?

```
hasNot ( Formula f ) {
    return f.accept (new HasNot ());
}
```

↑ defined as sep class where each method is for diff type

(7)

Seperate class for HasNot to seperate out cleanly
- but usually in same file

Can put class inside other method
- local class

Can get rid of name and ~~drop~~ drop entire class into return statement

```
return f.accept( new Visitor<Boolean>() {
        public Boolean onVar(Var v) { return false; }
        etc

});
```

But lots of objects created

Can make it a constant
Then just return this

Not really a performance problem
- Java is good at creating + destroying objects

(8)

Must add new functions to interface

Then add to each method

Java Gives a compile error unless you fully implement it
 ^ static checking

What if add. params to pass along to function

(missed)

~~Java~~

~~Visita~~ Env must be final

- not sure if it can auto copy refs

Very Complex use of Visitor

- don't try on P-set/Project

Same: Formula x Formula → boolean

same (V, V):

same (V, N):   (N, N) = same(~~...~~ n1.f, n2.f)

same (V, A):

- need to choose 1 to be reciever in 1st
- Visitor lets you do all 16

6.005 Software Construction
Fall 2011
Prof. Rob Miller

L8: Interpreters & Visitors

Today

- Design patterns
- Interpreter pattern
- Visitor pattern

design patterns

Required Reading (from the Java Tutorial & Thinking in Java)

- Inner classes
- Generic methods & constructors

Recipes for Program Construction

Let's take a moment to review the approaches we've been taking to building software in this course. Each of these is a recipe for tackling a different-sized software construction problem. In the first project, you'll have complete design freedom, so use it well.

Recipe for a method:

1. **Design.** Write the signature and specification. Think from the client's point of view. Common mistake is to put too much detail about how the implementation works into the spec. That reduces the implementer's freedom, and makes the spec harder to understand by itself.
2. **Test.** Write test cases that cover the spec.
3. **Code.** Implement the method. Make sure your test cases cover the code of the method (e.g. using a code coverage tool like EclEmma), and add more test cases until you achieve coverage.

Recipe for an abstract data type (a class of mutable or immutable objects):

1. **Design.**
    a. Choose the operations: the creators, producers, mutators, and observers that a client of the type will need.
        i. If the type is mutable, draw a state machine to define its important states and the events (mutators) that transition between them.
        ii. If the type involves recognizing or generating a structured language, write a grammar describing the language.
        iii. If the type is recursive, write a datatype expression showing how it's structured, and convert it into interfaces and classes.
    b. Write signatures and specifications for each of the operations.
2. **Test.**
    a. Write test cases that cover the transitions of the state machine, and cover the specs of the methods.
3. **Code.**
    a. Choose the rep, and write down the rep invariant and abstraction function.
    b. Implement the methods, running your test suite frequently as you go to check for regressions.
    c. Make sure your tests cover your code.

Finally, the recipe for a program:

These lecture notes have been collaboratively authored, with contributions from Saman Amarasinghe, Srini Devadas, Michael Ernst, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. Copyright © MIT.

1. **Choose data structures**. What data types will we use, and can we get them from a library or build them? Should the hailstone sequence be stored as a list or an array? For a lexer and parser, what do the tokens look like? Data is central to most programs. If you show me your data types, I can guess how your code works. If you just show me your code without your data types, I'll continue to be mystified.[1]
2. **Divide into steps**. For example, the steps of computing a hailstone sequence, then finding the peaks in it. Computing the digits of pi, converting to base-26, replacing with a-z, and searching for words. Lexing into tokens, then parsing into expressions.
3. **Module by module.** Implement each new data structure you'll need using the recipe for data types. Do it in isolation, as a unit. For example, Sudoku vs. Formula. Then implement each step of the process. Some of these steps will be methods in your data types; others may be static methods that stand alone in their own classes. Unit-test everything you can.
4. **Put it together**. Connect modules together, one by one, and test that they work together. Incremental is very important – an engineer expects things to fail, and makes only a small change at a time. Whenever you make a change or fix a bug at this stage, rerun *all* your tests to check for regressions.

Often you can follow these recipes straight through, but not always. You will sometimes discover when you're coding that your design was incomplete, and you need to back up and modify it, rewrite some tests. But if you don't start with a design – if you don't put some thought and preplanning into what you're doing -- then you'll be doing far more rewriting than necessary, and you are likely to end up with something unmaintainable – unsafe, hard to understand, and hard to change.

Functions over Recursive Data Types

Last time we looked briefly at one representation for formulas of boolean variables, which was defined recursively as:

Formula = Var(name:String)
+ Not(f: Formula)
+ And(left: Formula, right: Formula)
+ Or(left: Formula, right: Formula)

We can implement it as an interface Formula and four classes that implement it:

```
public static interface Formula {
}

public static class Var implements Formula{
    public final String name;
    public Var(String name) { this.name = name; }
}

public static class Not implements Formula{
```

[1] This is a paraphrase of a classic quote by Fred Brooks, who wrote the seminal book on software project management, *The Mythical Man-Month.* Brooks's original quote referred to flowcharts and tables, which were popular notions in the 1960s. Eric Raymond, who wrote the book on open source software development, *The Cathedral and the Bazaar,* paraphrased his quote into data structures and code, for the C programmers of the 90s. Guy Steele, who wrote the book on Lisp and co-invented Java, paraphrased it again into contracts and code, for the new millenium. http://dreamsongs.com/ObjectsHaveNotFailedNarr.html

```
        public final Formula f;
        public Not(Formula f) { this.f = f; }
    }

    public static class And implements Formula{
        public final Formula left, right;
        public And(Formula left, Formula right) { this.left = left; this.right
= right; }
    }

    public static class Or implements Formula{
        public final Formula left, right;
        public Or(Formula left, Formula right) { this.left = left; this.right
= right; }
    }
```

Last time we also saw that we could define recursive functions over datatypes, such as:

eval: Formula x Env → boolean

eval(Var(name:String), env) = env.lookup(name)

eval(Not(f:Formula) , env) = ¬ eval(f, env)

eval(And(left, right: Formula) , env) = eval(left, env) ∧ eval(right, env)

eval(Or(left, right: Formula) , env) = eval(left, env) ∨ eval(right, env)

which translates directly into an eval( ) method implemented on the variant classes:

```
    public static interface Formula {
        public boolean eval(Env env);
    }

    public static class Var implements Formula {
        ...
        public boolean eval(Env env) { return env.lookup(name); }
    }

    public static class Not implements Formula {
        ...
        public boolean eval(Env env) { return !f.eval(env); }
    }

    public static class And implements Formula {
        ...
        public boolean eval(Env env) { return left.eval(env) &&
right.eval(env); }
    }

    public static class Or implements Formula {
        ...
        public boolean eval(Env env) { return left.eval(env) ||
right.eval(env); }
    }
```

This is a great approach for defining core operations of the datatype, like eval().

But this approach won't work for all the functions we might want. Suppose some client wants a hasNot() function that tests whether there's any negation in the formula:

> hasNot: Formula → boolean
>
> > hasNot(Var(name:String)) = false
> >
> > hasNot(Not(f:Formula)) = true
> >
> > hasNot(And(left, right: Formula)) = hasNot(left) || hasNot(right)
> >
> > hasNot(Or(left, right: Formula)) = hasNot(left) || hasNot(right)

Another might want a countVars() function that counts the variables, or a varSet() function that collects all the variables, or a hasLowercaseVariable() that looks for a particular kind of variable name that presumably has semantic meaning for the client... We can't fill up our Formula interface with methods needed by only one client; that would destroy its coherence as an abstract data type.

A final problem with this approach is that it spreads out the code that implements the function into multiple classes – in fact, multiple *files* in a language like Java. That makes it much harder to understand, because a programmer has to piece together the way eval() works by looking in multiple places, and make sure that those places are kept consistent with each other when the function is changed. It doesn't show everything about eval() in one place, like our mathematical function definition does.

Design Patterns

The recursive-function-as-method approach has a standard name in software engineering: it's the **Interpreter** design pattern. We'll see another approach in a moment, but first a word about the notion of design patterns.

A design pattern is a standard solution to a common programming problem.

Design patterns are like standard tools in your toolbox. When you see a common problem, you can turn to your collection of design patterns and decide whether one fits the situation or not.

Design patterns are also a useful **vocabulary** for talking about software design. You can tell another programmer to "throw an *exception*" or "use a *factory method*" or "create a *visitor*," and they will know what you are talking about. So design patterns are useful abstractions for communicating the meaning of your program to other humans, too, not just for communicating with the compiler.

We've already encountered some important design patterns in this course:

Encapsulation (aka information hiding)

Problem: Exposed fields can be directly manipulated from outside, leading to violations of the representation invariant or undesirable dependences that prevent changing the implementation.
Solution: Hide some components, permitting only stylized access to the object – public getX() methods rather than direct access to a private field x.
Disadvantages: The interface may not (efficiently) provide all desired operations. Indirection may reduce performance.

Iterator

Problem: Clients that wish to access all members of a collection must perform a specialized traversal for each data structure. This introduces undesirable dependences and does not extend to other collections.

Solution: Implementations, which have knowledge of the representation, perform traversals and do bookkeeping. The results are communicated to clients via a standard interface (called Iterator in Java).
Disadvantages: Iteration order is fixed by the implementation and not under the control of the client. You can't go backwards through the elements, or jump around in a pseudorandom sequence.

Exceptions

Problem: Errors occurring in one part of the code should often be handled elsewhere. Code should not be cluttered with error-handling code, nor return values preempted by error codes.
Solution: Introduce language structures for throwing and catching exceptions.
Disadvantages: Code may still be cluttered. It can be hard to know where an exception will be handled. Programmers may be tempted to use exceptions for normal control flow, which is confusing and usually inefficient.

These particular design patterns are so important that they are built into Java! Other design patterns are so important that they are built into other languages. Some design patterns may never be built into languages, but are still useful in their place. Other patterns we've thought about include **state machines**, **lexer/parser**, and **abstract syntax tree**.

Factory Pattern

Here's a design pattern that you'll see frequently with abstract data types, particularly in languages like Java that restrict what interfaces and constructors can do: the **Factory Method** pattern.

Recall that abstract data types have several different kinds of operations: creators, producers, observers, and mutators. When we first talked about ADTs, we equated a creator with a constructor. But that's not really enough for abstract data types that want to hide their concrete classes. Remember ImList:

```
public static interface ImList<E> {
    public ImList<E> add(E e);
    public E first();
    public ImList<E> rest();
}

public static class Empty<E> implements ImList<E> {
    public Empty() { }
    ...
}
```

We want a client to be able to make an empty list without having to know about the Empty class, so that ImList has the freedom to change it in the future (representation independence). Unfortunately we can just put a constructor in the ImList interface:

```
public static interface ImList<E> {
    /** make an empty list */
    public ImList<E>() { ... }  // <==== Java forbids constructors in
interfaces
```

A factory method lets us handle this problem. It's simply a static method that creates a value of a type:

```
/** @return an empty list */
public <E> ImList<E> empty() {
```

```
        return new Empty();
    }
```

Here's another useful factory method for ImList:

```
    /** @param lst  a mutable java.util.List
     *  @return a new immutable ImList representing the same sequence as lst */
    public <E> ImList<E> makeList(List<E> lst) { ... }
```

Factory methods simplify client code and provide a layer of encapsulation between the client and the implementer.

Its' so simple, I don't get it!

Patterns for Functions over Recursive Datatypes

Interpreter and **Visitor** are the major design patterns we'll see today. We've actually already had a glimpse of Interpreter, which is we were calling the recursive-function-as-method approach. Java has good support for it with interface methods and implementation methods.

Visitor is actually built into some functional programming languages, as pattern matching in ML and Haskell. But it's not built into Java, and it turns out to be very useful for implementing functions over recursive datatypes like lists and trees.

Back to our problem of how to represent a function over a recursive datatype, without adding a new method to the datatype's interface. Let's consider the hasNot function:

hasNot: Formula → boolean

hasNot(Var(name:String)) = false

hasNot(Not(f:Formula)) = true

hasNot(And(left, right: Formula)) = hasNot(left) || hasNot(right)

hasNot(Or(left, right: Formula)) = hasNot(left) || hasNot(right)

The key idea of Visitor is to bring all of these cases into one place, so we wish we could do something like this:

```
public static boolean hasNot(Formula f) {
    // not valid Java code!
    switch (f) {
    case Var v: return false;
    case Not n: return true;
    case And a: return hasNot(a.left) || hasNot(a.right);
    case Or o:  return hasNot(o.left) || hasNot(o.right);
    }
}
```

based on type

The switch statement in Java doesn't work that way – this would be the right thing to do if f were an enum, but it doesn't work for object types. Here's another try that is legal Java but very bad style:

```
public static boolean hasNot(Formula f) {
    // not statically checked!
    if (f instanceof Var) {
        return false;
    } else if (f instanceof Not) {
        return true;
    } else if (f instanceof And) {
        And a = (And) f;
        return hasNot(a.left) || hasNot(a.right);
    } else if (f instanceof Or) {
        Or o = (Or) f;
        return hasNot(o.left) || hasNot(o.right);
    } else {
        throw new AssertionError("shouldn't get here!");
    }
}
```

What this code does is determine the type of the list using instanceof (see the Java Tutorial for more details), and then *downcasts* from the interface type Formula to the concrete class types so that it can use their specific fields and methods.

The code above is bad because it gives up static checking. One way that datatypes often change is by adding new variants, new concrete classes that implement Formula. Suppose somebody extends the boolean formulas to support existential and universal quantification -- ThereExists and ForAll. If that happened, the compiler would *force* the maintainer to implement the eval() method for these new variants because it was mentioned in the Formula interface. The program wouldn't even compile unless the maintainer took care of eval(), so functions implemented with the Interpreter pattern are statically checked. But hasNot() written as shown above would continue to compile, happily but wrongly, with no static checking to remind us that we need to change that function too.

The code above does do one thing right: at least it defends itself against unexpected new variants of Formula that it encounters at runtime. It doesn't assume that Var, Not, And, and Or are the only possible implementations of Formula, and it throws an error if it discovers a new one, rather than returning wrong answers. The following code would be much much worse:

```
public static boolean hasNot(Formula f) {
    if (f instanceof Var) {
        return false;
    } else if (f instanceof And) {
        And a = (And) f;
        return hasNot(a.left) || hasNot(a.right);
    } else if (f instanceof Or) {
        Or o = (Or) f;
        return hasNot(o.left) || hasNot(o.right);
    } else {
        // must be a Not     <=== not defensive!
        return true;
    }
}
```

Can't tell an error occured

I just can't put enough big red X's on this code.

In general, instanceof should be avoided. There is almost always a way to write your code that doesn't require testing the type of an object with instanceof and downcasting. Use of instanceof is a sign of a bad smell. There are exceptions to this – instanceof is frequently used to implement equals(), as we'll see – but generally instanceof should not be part of a good object-oriented design.

2nd shown in lecture

The Visitor Pattern

We saw two almost-right approaches. Here's how the visitor pattern actually works. We'll follow the model of the switch statement we wished we could write, and show how each part of it can be represented in correct, typesafe, statically-checked Java.

Let's start by thinking about the body of our imaginary switch statement, which has the four cases for the function. To represent that, we'll define a **visitor class** with four methods, one for each variant:

Method has its own Class!

```
class HasNot implements Visitor<Boolean> {
    public Boolean onVar(Var v) { return false; }
    public Boolean onNot(Not n) { return true; }
    public Boolean onAnd(And a) { return hasNot(a.left) ||
hasNot(a.right); }
    public Boolean onOr(Or o) { return hasNot(o.left) || hasNot(o.right);
}
  }
```

Override based on type

Note the Visitor interface that this class implements. This interface describes the general pattern for writing cases of recursive functions over the variants of Formula:

```
public interface Visitor<R> {
    public R onVar(Var v);
    public R onNot(Not n);
    public R onAnd(And a);
    public R onOr(Or o);
}
```

define in interface. all possible types that each method has to implement

It's parameterized by type R, which is the return value of the function you're defining. hasNot() returns a boolean, so HasNot implements Visitor<Boolean>.

Now we need a mechanism for invoking this visitor class – something that switches on the actual type of the formula object and calls the appropriate case in our visitor object. The trick here is to use the Interpreter pattern to invoke the visitor, by adding a new method to the datatype, conventionally called *accept*:

is using both

```
public interface Formula {
    ...
    public <R> R accept(Visitor<R> v);
}

public class Var implements Formula{
    ...
    public <R> R accept(Visitor<R> v) { return v.onVar(this); }
}

public class Not implements Formula{
    ...
    public <R> R accept(Visitor<R> v) { return v.onNot(this); }
}

public class And implements Formula{
    ...
```

So class for each type as well

- lots of classes...
- must you put each in own file

```
    public <R> R accept(Visitor<R> v) { return v.onAnd(this); }
}

public class Or implements Formula{
    ...
    public <R> R accept(Visitor<R> v) { return v.onOr(this); }
}
```

What is accept?

Study accept() closely, because some magic is happening there. First a minor piece of Java syntax: in order to allow accept() to handle visitors of any return type, we need to parameterize the entire accept() method by the type variable R, so <R> appears at the start of the signature. But the key magic of the visitor pattern is what happens in the body of the method. Each case of the accept function chooses a different method of the visitor object to invoke: onVar, onNot, onAnd, onOr. And it always passes along "this," which has a declared type corresponding to the concrete class in which it appears (Var, Not, And, or Or, respectively), rather than the abstract type Formula. So the Java compiler can look at the onVar call inside Var and say – yes! this is guaranteed to be a Var here, so it's safe for you to pass it here. We have static type checking.

Finally, we put all the pieces together with a static method hasNot:

```
public static boolean hasNot(Formula f) {
    return f.accept(new HasNot());
}
```

Here, we make the visitor object that contains the code for the cases of the hasNot function, and then dispatch to it through accept(). Walk through the code for a case like

And(Not(Var("x")), Var("y"))

to make sure you understand how it works. Draw a snapshot diagram of the formula first, so that you can keep track of what's going on.

So method hasNot()
returns f.accept(new HasNot)
accept defined in each data type
returns the type
to accept interface
implemented by actual fn
returns something diff based on type
(makes my head spin - no simpler way to do?
- well I guess if want static checking

A few tweaks

Here are a few ways to tune the visitor pattern to make it more readable in Java:

- Since we may have several recursive data types that need to use the visitor pattern, it helps to nest the Visitor interface inside the datatype that needs it, in this case Formula.
- We can exploit Java's method overloading – using the same name for different methods that take different types – to change onVar, onNot, etc. to all use the same name, "on".
- Finally, we don't have to declare the HasNot visitor as a named class; we can use Java's anonymous class syntax instead, and put it right inside the hasNot() method, the only place where it's used.

Here's the result:

```
public interface Formula {
    public interface Visitor<R> {
        public R on(Var v);
        public R on(Not n);
        public R on(And a);
        public R on(Or o);
    }
    public <R> R accept(Visitor<R> v);
}
...
public static boolean hasNot(Formula f) {
    return f.accept(new Formula.Visitor<Boolean>() {
        public Boolean on(Var v) { return false; }
        public Boolean on(Not n) { return true; }
        public Boolean on(And a) { return hasNot(a.left)
                                          || hasNot(a.right); }
        public Boolean on(Or o) { return hasNot(o.left
                                          || hasNot(o.right); }
    });
}
```

There's also a performance tweak. If you trace through a invocation of hasNot() on a formula tree, you'll notice that every time hasNot() is called recursively, it creates a new visitor object, which then goes away when hasNot() returns. Since all these objects are identical and immutable, this is unnecessary overhead that can be eliminated simply by creating the object once and reusing it:

```
public static boolean hasNot(Formula f) {
    return f.accept(HAS_NOT_VISITOR);
}
private static Formula.Visitor<Boolean> HAS_NOT_VISITOR
  = new Formula.Visitor<Boolean>() {
    public Boolean on(Var v) { return false; }
    public Boolean on(Not n) { return true; }
    public Boolean on(And a) { return hasNot(a.left) || hasNot(a.right); }
    public Boolean on(Or o) { return hasNot(o.left) || hasNot(o.right); }
};
```

If hasNot() is used rarely, however, the cost in readability may not be worth the performance benefit. This performance tweak is also incompatible with having multiple arguments to the function, as we'll see next.

Passing parameters to a visitor

hasNot() is a function with only one argument: the Formula. What if we need additional arguments, like eval(Formula, Env)? We can make these additional arguments available to the cases of the function by storing them as fields of the visitor object:

```
    public static boolean eval(Formula f, Env env) {
        return f.accept(new Eval(env));
    }

    static class Eval implements Formula.Visitor<Boolean> {
        private Env env;
        public Eval(Env env) { this.env = env; }
        public Boolean on(Var v) { return env.lookup(v.name); }
        public Boolean on(Not n) { return !eval(n.f, env); }
        public Boolean on(And a) { return eval(a.left, env) && eval(a.right,
env); }
        public Boolean on(Or o)  { return eval(o.left, env) || eval(o.right,
env); }
    }
```

In Java, we can also express this visitor as an anonymous class:

```
    public static boolean eval(Formula f, final Env env) {
        return f.accept(new Formula.Visitor<Boolean>() {
            public Boolean on(Var v) { return env.lookup(v.name); }
            public Boolean on(Not n) { return !eval(n.f, env); }
            public Boolean on(And a) { return eval(a.left, env)
                                           && eval(a.right, env); }
            public Boolean on(Or o)  { return eval(o.left, env)
                                           || eval(o.right, env); }

        });
    }
```

Key to note here is that any local variables declared *final* are automatically in scope for the anonymous class. So we didn't have to do anything special to make env available to the cases of the visitor; Java handles all that automatically.

Different Perspectives on Visitor

Visitor as a kind of type switch

Compare our final code for hasNot() with the original switch statement we wanted to write – the structure is identical. Essentially we have implemented our own type dispatch – a mechanism for looking at the actual type of an object and choosing which piece of code to run in response. That's what a switch statement does for an enumerated type; that's what calling a method through an interface ordinarily does for methods defined on the interface itself; what we've done with visitor is created our own version of that, which allows the pieces of code to be stored in a separate object.

There are other ways we can use this type-dispatch mechanism as well. For example, we can use it for *multiple dispatch* -- choosing a piece of code to run based on the types of more than one argument. Suppose you have a function that takes *two* Formulas as arguments – can we dispatch all 4x4 = 16

cases of this function, without using instanceof? Yes we can. Fasten your seatbelts, let's implement a function that tests whether two formulas are syntactically identical.

same: Formula x Formula -> boolean

same(Var(name1:String), Var(name2:String)) = (name1 = name2)

same(Var(name1:String), Not(formula2:Formula)) = false

...

same(Not(formula1:Formula), Var(name2:String)) = false

same(Not(formula1:Formula), Not(formula2:Formula)) = same(formula1, formula2)

...

We've only shown four cases here, the other 12 are similar. The code for this starts out like this:

```
public static boolean same(final Formula f1, final Formula f2) {
    return f1.accept(new Formula.Visitor<Boolean>() {
        public Boolean on(final Var v1) {
            return f2.accept(new Formula.Visitor<Boolean>() {
                public Boolean on(Var v2) { return v1.name
                                              .equals(v2.name); }
                public Boolean on(Not n2) { return false; }
                public Boolean on(And a2) { return false; }
                public Boolean on(Or o2)  { return false; }
...
```

Frankly multiple dispatch is rarely needed in practice (and equality tests tend to be written with instanceof, because they can't assume that both objects are Formulas to begin with). But the visitor pattern can do it.

Visitor as a function over a datatype

Our original reason for the Visitor pattern in this lecture was to solve the problem of representing a function over a recursive datatype. But one interesting feature to call out here is that in the final pattern, we have an *object* that represents a function. A visitor object is not a state machine, it's not a value of an abstract data type, it's a new kind of beast – an object that is essentially represents a method. Like all objects, this method-as-an-object is *first class*, which means that we can name it with a variable, pass it around, store it in data structures, and return it from other methods:

hasnot = new HasNotVisitor();

To actually call the visitor, we hand it to accept(). Notice the minor difference in syntax:

f.eval(); // if eval uses the interpreter pattern

f.accept(hasnot); // if hasNot uses the visitor pattern

But where hasnot is a reference to a first-class object, eval is just a method name. We can't store a reference to it or treat it like data.

Visitor as a kind of iterator

You may have noticed that the visitor pattern for Formula effectively **iterates** over the formula tree, visiting each node. (That's why it's called the visitor pattern, in fact.)

Compare an iterator over a collection:

```
// compute the sum of a list
List<Integer> lst = ...;
int sum = 0;
for (int x: lst) {
    sum = sum + x;
}
```

with a visitor over a recursive data type:

```
// count the variables in a formula
Formula f = ...;
f.accept(new Formula.Visitor<Integer> () {   // aka "for each node..."
    public Integer on(Var v) { return 1; }
    public Integer on(Not n) { return n.f.accept(this); }
    public Integer on(And a) { return a.left.accept(this)
                                + a.right.accept(this); }
    public Integer on(Or o)  { return o.left.accept(this)
                                + o.right.accept(this); }
}
```

The iteration steps through a collection, iteratively assigning the variable *x* to different elements, and executing the body of the *for* loop once for each element. The visitor pattern steps through the formula tree, assigning one of the variables v, n, a, or o to each node (depending on its type), and executing the appropriate case for that type.

Unlike Iterator, which basically steps through all the elements in the collection in an order that's up to the implementer of the collection, the Visitor pattern gives the client more control over navigation. Our hasNot visitor was able to choose not even to drill down into Not's subformula (since it doesn't affect the final answer), and its And/Or cases could look at their left or right subtrees in either order.

Visitors aren't limited to pure functions with no side-effects or local state. A visitor object can be a state machine that updates itself as it traverses the tree, in the similar way that the for loop updates the sum variable as it steps through the collection. Consider an operation that finds all the variables in a formula and accumulates them in a Set<String>:

```
public static Set<String> varSet (Formula f) {
    VarSet vs = new VarSet();
    f.accept(vs);
    return vs.vars;
}

private class VarSet implements Formula.Visitor<Void> {
    public final Set<String> vars = new HashSet<String>();
    public Void on(Var v) { vars.add(v.name); return null; }
    public Void on(Not n) { n.f.accept(this); return null; }
    public Void on(And a) { a.left.accept(this); a.right.accept(this);
                            return null; }
    public Void on(Or o)  { o.left.accept(this); o.right.accept(this);
                            return null; }
}
```

VarSet is a state machine whose transitions are the four on() methods. Two things to clarify here: first, since the on methods have become mutators, we don't care about their return value anymore,

so we use the Java builtin class Void (which is the object type that corresponds to the void primitive type). This type has no values, but Java still requires a return value, so we return null. Second, the recursive calls in the cases for Not, And, and Or want to use the *same* visitor object for the recursive part of the traversal, so we can't just call varSet() recursively. Instead we directly apply the visitor object (this!) to the subtree, using its accept method.

Here's a more compact way to write the same operation, using an anonymous class so that all the code is contained within varSet():

```
public static Set<String> varSet (Formula f) {
    final Set<String> vars = new HashSet<String>();
    f.accept(new Formula.Visitor<Void>() {
        public Void on(Var v) { vars.add(v.name); return null; }
        public Void on(Not n) { n.f.accept(this); return null; }
        public Void on(And a) { a.left.accept(this); a.right.accept(this);
                                return null; }
        public Void on(Or o)  { o.left.accept(this); o.right.accept(this);
                                return null; }
    });
    return vars;
}
```

Dual nature of Interpreter and Visitor

Interpreter and Visitor are duals of each other, in the following sense. Let's think about a recursive type in terms of its operations (the rows) and its variant classes (the columns). Each operation needs a piece of code to handle each variant, so each cell in this table is essentially a method body somewhere.

	Var	Not	And	Or	*ThereExists?*	*ForAll?*
eval						
hasNot						
same						
varSet						
toCNF?						

Interpreter is column-centric: it groups all the code for a variant together in one place, inside the variant class. Visitor is row-centric: it groups all the code for a single operation together in one place, inside the visitor class.

Interpreter makes it easy to add a new variant – you just create a new variant class with all the cases for that variant in it -- but hard to add a new operation, because you'll be touching every single variant class to do it. Every variant has to have a complete list of the operations that use the Interpreter pattern.

Visitor, by contrast, makes it easy to add a new operation, as a new visitor class, but hard to add a new variant. Every visitor has to have a complete list of all the variants.

So one design tradeoff between these two patterns boils down to **which kind of change you expect** in the future. Is the set of variants pretty fixed, but clients are likely to want to invent new

that is why we are doing all this I believe

operations? Then you want visitors. Or are clients likely to need new variants – maybe even variants that the client creates themselves? Then you want interpreters.

Interpreter: function as methods

- Has privileged access to reps of the variant classes
- Harder to understand the function, because it's split over multiple files
- Adding a new function requires changing every variant class
- Use for operations that are essential to the abstract data type

Visitor: function as visitor object

- Reduces rep independence: variants must be visible to client
- Need observer operations to get parts of the rep
- Function is found all in one place, so is easier to understand & change
- Adding a new variant requires changing every visitor class
- Use for operations that are essential to some client's processing

Both patterns are statically checked, though!

Oh - so both can work depending on use case

Summary

We've looked at the notion of **design patterns**, which are standard solutions to common design problems. Some design patterns are formalized in programming languages, and some need to be brought in by a programmer. Design patterns have names that are worth committing to memory (factory method, interpreter, visitor) so that you can communicate with other programmers.

We've also delved deeply into the pros and cons of two design patterns for recursive data types, interpreter and visitor. Interpreter is well-supported by Java language features; visitor requires us to implement more of the machinery ourselves. But the benefits that you get from a visitor, in the form of pulling together the code for a single operation in one place, with full static type checking, are extremely powerful.

```
package inclass;


public class Formula1_datatype {
    // datatype definition:
    //    Formula = Var(name:String)
    //              + Not(f:Formula)
    //              + And(left:Formula, right:Formula)
    //              + Or(left:Formula, right:Formula)

    public static interface Formula {
    }

    public static class Var implements Formula{
        public final String name;
        public Var(String name) { this.name = name; }
    }
    public static class Not implements Formula{
        public final Formula f;
        public Not(Formula f) { this.f = f; }
    }
    public static class And implements Formula{
        public final Formula left, right;
        public And(Formula left, Formula right) { this.left = left; this.right = right; }
    }
    public static class Or implements Formula{
        public final Formula left, right;
        public Or(Formula left, Formula right) { this.left = left; this.right = right; }
    }
}
```

¿So what are each of these?

Interpreter Pattern

Good

```
package inclass;

public class Formula2_interpreter_pattern {
    // datatype definition:
    //    Formula = Var(name:String)
    //              + Not(f:Formula)
    //              + And(left:Formula, right:Formula)
    //              + Or(left:Formula, right:Formula)

    public static interface Formula {
        public boolean eval(Env env);
    }
    public static class Var implements Formula {
        public final String name;
        public Var(String name) { this.name = name; }    ← constructor
        public boolean eval(Env env) { return env.lookup(name); }
    }                                         ↑ sep defined

    public static class Not implements Formula {
        public final Formula f;
        public Not(Formula f) { this.f = f; }
        public boolean eval(Env env) { return !f.eval(env); }
    }

    public static class And implements Formula {
        public final Formula left, right;             recursive
        public And(Formula left, Formula right) { this.left = left; this.right = right; }
        public boolean eval(Env env) { return left.eval(env) && right.eval(env); }
    }

    public static class Or implements Formula {
        public final Formula left, right;
        public Or(Formula left, Formula right) { this.left = left; this.right = right; }
        public boolean eval(Env env) { return left.eval(env) || right.eval(env); }
    }


    public static interface Env {
        public boolean lookup(String name);
    }
}
```

```
package inclass;

import inclass.Formula2_interpreter_pattern.Formula;

public class Formula3_instanceof {
    // datatype definition:
    //    Formula = Var(name:String)
    //              + Not(f:Formula)
    //              + And(left:Formula, right:Formula)
    //              + Or(left:Formula, right:Formula)

    public static interface Formula {
    }

    public static class ThereExists implements Formula {
    }

    public static class Var implements Formula{
        public final String name;
        public Var(String name) { this.name = name; }
    }

    public static class Not implements Formula{
        public final Formula f;
        public Not(Formula f) { this.f = f; }
    }

    public static class And implements Formula{
        public final Formula left, right;
        public And(Formula left, Formula right) { this.left = left; this.right = right; }
    }

    public static class Or implements Formula{
        public final Formula left, right;
        public Or(Formula left, Formula right) { this.left = left; this.right = right; }
    }

    public static boolean hasNot(Formula f) {
        // bad idea, not statically checked!
        if (f instanceof Var) {
            return false;
        } else if (f instanceof Not) {
            return true;
        } else if (f instanceof And) {
            And a = (And) f;
            return hasNot(a.left) || hasNot(a.right);
        } else if (f instanceof Or) {
            Or o = (Or) f;
            return hasNot(o.left) || hasNot(o.right);
        } else {
            throw new AssertionError("shouldn't get here!");
        }
```

Instance Of Bad

adding user defined methods?

```java
package inclass;

public class Formula4_really_bad_instanceof {
    // datatype definition:
    //    Formula = Var(name:String)
    //              + Not(f:Formula)
    //              + And(left:Formula, right:Formula)
    //              + Or(left:Formula, right:Formula)

    public static interface Formula {
    }

    public static class Var implements Formula{
        public final String name;
        public Var(String name) { this.name = name; }
    }

    public static class Not implements Formula{
        public final Formula f;
        public Not(Formula f) { this.f = f; }
    }

    public static class And implements Formula{
        public final Formula left, right;
        public And(Formula left, Formula right) { this.left = left; this.right = right; }
    }

    public static class Or implements Formula{
        public final Formula left, right;
        public Or(Formula left, Formula right) { this.left = left; this.right = right; }
    }

    public static boolean hasNot(Formula f) {
        if (f instanceof Var) {
            return false;
        } else if (f instanceof And) {
            And a = (And) f;
            return hasNot(a.left) || hasNot(a.right);
        } else if (f instanceof Or) {
            Or o = (Or) f;
            return hasNot(o.left) || hasNot(o.right);
        } else {
            // must be a Not    <=== not defensive!
            return true;
        }
    }

}
```

Instance Of Bad

← looks valid

Visitor Pattern

Good

```
package inclass;

public class Formula5_visitor_pattern {
    // datatype definition:
    //    Formula = Var(name:String)
    //              + Not(f:Formula)
    //              + And(left:Formula, right:Formula)
    //              + Or(left:Formula, right:Formula)

    public static interface Formula {
        // the method that invokes a visitor object
        public <R> R accept(Visitor<R> v);
    }

    public static class ThereExists implements Formula {
        public final Formula f;
        public ThereExists(Formula f) { this.f = f; }
        public <R> R accept(Visitor<R> v) { return v.onThereExists(this); }

    }

    public static class Var implements Formula{
        public final String name;
        public Var(String name) { this.name = name; }
        public <R> R accept(Visitor<R> v) { return v.onVar(this); }
    }

    public static class Not implements Formula{
        public final Formula f;
        public Not(Formula f) { this.f = f; }
        public <R> R accept(Visitor<R> v) { return v.onNot(this); }
    }

    public static class And implements Formula{
        public final Formula left, right;
        public And(Formula left, Formula right) { this.left = left; this.right = right; }
        public <R> R accept(Visitor<R> v) { return v.onAnd(this); }
    }

    public static class Or implements Formula{
        public final Formula left, right;
        public Or(Formula left, Formula right) { this.left = left; this.right = right; }
        public <R> R accept(Visitor<R> v) { return v.onOr(this); }
    }

    // the interface that all visitor classes implement
    public interface Visitor<R> {
        public R onVar(Var v);
        public R onNot(Not n);
        public R onAnd(And a);
        public R onOr(Or o);
        public R onThereExists(ThereExists e);
```

```
    }
                    ✓ add methods
    private static final Visitor<Boolean> HAS_NOT = new Visitor<Boolean>() {
        public Boolean onVar(Var v) { return false; }
        public Boolean onNot(Not n) { return true; }
        public Boolean onAnd(And a) { return hasNot(a.left) || hasNot(a.right); }
        public Boolean onOr(Or o) { return hasNot(o.left) || hasNot(o.right); }
        public Boolean onThereExists(ThereExists e) { return hasNot(e.f); }
    };
                              add types

    // the hasNot function
    public static boolean hasNot(Formula f) {
        return f.accept(HAS_NOT);
    }

    // the visitor class that has the cases of the function

}
```

```java
package inclass;


public class Formula6_tweaks_on_visitor {
    // datatype definition:
    //    Formula = Var(name:String)
    //              + Not(f:Formula)
    //              + And(left:Formula, right:Formula)
    //              + Or(left:Formula, right:Formula)

    public static interface Formula {
        public interface Visitor<R> {
            public R on(Var v);
            public R on(Not n);
            public R on(And a);
            public R on(Or o);
        }
        public <R> R accept(Visitor<R> v);
    }

    public static class Var implements Formula{
        public final String name;
        public Var(String name) { this.name = name; }
        public <R> R accept(Visitor<R> v) { return v.on(this); }
    }

    public static class Not implements Formula{
        public final Formula f;
        public Not(Formula f) { this.f = f; }
        public <R> R accept(Visitor<R> v) { return v.on(this); }
    }

    public static class And implements Formula{
        public final Formula left, right;
        public And(Formula left, Formula right) { this.left = left; this.right = right; }
        public <R> R accept(Visitor<R> v) { return v.on(this); }
    }

    public static class Or implements Formula{
        public final Formula left, right;
        public Or(Formula left, Formula right) { this.left = left; this.right = right; }
        public <R> R accept(Visitor<R> v) { return v.on(this); }
    }

    // using an anonymous class for a visitor
    public static boolean hasNot(Formula f) {
        return f.accept(new Formula.Visitor<Boolean>() {      ← shorter
            public Boolean on(Var v) { return false; }
            public Boolean on(Not n) { return true; }
            public Boolean on(And a) { return hasNot(a.left) || hasNot(a.right); }
            public Boolean on(Or o) { return hasNot(o.left) || hasNot(o.right); }
        });
```

```
    }


    // passing additional parameters to a visitor
    public static boolean eval(Formula f, Env env) {
        return f.accept(new Eval(env));
    }

    static class Eval implements Formula.Visitor<Boolean> {
        private Env env;
        public Eval(Env env) { this.env = env; }
        public Boolean on(Var v) { return env.lookup(v.name); }
        public Boolean on(Not n) { return !eval(n.f, env); }
        public Boolean on(And a) { return eval(a.left, env) && eval(a.right, env); }
        public Boolean on(Or o)  { return eval(o.left, env) || eval(o.right, env); }
    }

    // eval #2: putting the class inside the method
    public static boolean eval2(Formula f, Env env) {
        class Eval2 implements Formula.Visitor<Boolean> { local
            private Env env;                                 - even shorter
            public Eval2(Env env) { this.env = env; }
            public Boolean on(Var v) { return env.lookup(v.name); }
            public Boolean on(Not n) { return !eval(n.f, env); }
            public Boolean on(And a) { return eval(a.left, env) && eval(a.right, env); }
            public Boolean on(Or o)  { return eval(o.left, env) || eval(o.right, env); }

        }
        return f.accept(new Eval2(env));
    }

    // eval #3: using an anonymous class and final variables
    public static boolean eval3(Formula f, final Env env) {
        return f.accept(new Formula.Visitor<Boolean>() {
            public Boolean on(Var v) { return env.lookup(v.name); }
            public Boolean on(Not n) { return !eval(n.f, env); }
            public Boolean on(And a) { return eval(a.left, env) && eval(a.right, env); }
            public Boolean on(Or o)  { return eval(o.left, env) || eval(o.right, env); }

        });
    }
                        what is the difference?


    public static interface Env {
        public boolean lookup(String name);
    }
}
```

```
package inclass;

public class Formula7_multiple_dispatch {
    // datatype definition:
    //    Formula = Var(name:String)
    //              + Not(f:Formula)
    //              + And(left:Formula, right:Formula)
    //              + Or(left:Formula, right:Formula)

    public static interface Formula {
        public interface Visitor<R> {
            public R on(Var v);
            public R on(Not n);
            public R on(And a);
            public R on(Or o);
        }
        public <R> R accept(Visitor<R> v);
    }

    public static class Var implements Formula{
        public final String name;
        public Var(String name) { this.name = name; }
        public <R> R accept(Visitor<R> v) { return v.on(this); }
    }

    public static class Not implements Formula{
        public final Formula f;
        public Not(Formula f) { this.f = f; }
        public <R> R accept(Visitor<R> v) { return v.on(this); }
    }

    public static class And implements Formula{
        public final Formula left, right;
        public And(Formula left, Formula right) { this.left = left; this.right = right; }
        public <R> R accept(Visitor<R> v) { return v.on(this); }
    }

    public static class Or implements Formula{
        public final Formula left, right;
        public Or(Formula left, Formula right) { this.left = left; this.right = right; }
        public <R> R accept(Visitor<R> v) { return v.on(this); }
    }


    /**
     * @return true if f1 and f2 are syntactically identical
     */
    // Demonstrates multiple dispatch using visitors.  Rarely used!  Not often
    // worth the complexity in practice, but worth understanding how it works.
    public static boolean same(final Formula f1, final Formula f2) {
        return f1.accept(new Formula.Visitor<Boolean>() {
            public Boolean on(final Var v1) {
```

```
                return f2.accept(new Formula.Visitor<Boolean>() {
                    public Boolean on(Var v2) { return v1.name.equals(v2.name); }
                    public Boolean on(Not n2) { return false; }
                    public Boolean on(And a2) { return false; }
                    public Boolean on(Or o2)  { return false; }
                });
            }
            public Boolean on(final Not n1) {
                return f2.accept(new Formula.Visitor<Boolean>() {
                    public Boolean on(Var v2) { return false; }
                    public Boolean on(Not n2) { return same(n1.f, n2.f); }
                    public Boolean on(And a2) { return false; }
                    public Boolean on(Or o2)  { return false; }
                });
            }
            public Boolean on(final And a1) {
                return f2.accept(new Formula.Visitor<Boolean>() {
                    public Boolean on(Var v2) { return false; }
                    public Boolean on(Not n2) { return false; }
                    public Boolean on(And a2) { return same(a1.left, a2.left) && same(a1.
right, a2.right); }
                    public Boolean on(Or o2)  { return false; }
                });
            }
            public Boolean on(final Or o1) {
                return f2.accept(new Formula.Visitor<Boolean>() {
                    public Boolean on(Var v2) { return false; }
                    public Boolean on(Not n2) { return false; }
                    public Boolean on(And a2) { return false; }
                    public Boolean on(Or o2)  { return same(o1.left, o2.left) && same(o1.
right, o2.right); }
                });
            }
        });
    }
}
```

all that depth

```java
package inclass;

import java.util.HashSet;
import java.util.Set;

public class Formula8_mutable_visitor {
    // datatype definition:
    //    Formula = Var(name:String)
    //              + Not(f:Formula)
    //              + And(left:Formula, right:Formula)
    //              + Or(left:Formula, right:Formula)

    public static interface Formula {
        public interface Visitor<R> {
            public R on(Var v);
            public R on(Not n);
            public R on(And a);
            public R on(Or o);
        }
        public <R> R accept(Visitor<R> v);
    }

    public static class Var implements Formula{
        public final String name;
        public Var(String name) { this.name = name; }
        public <R> R accept(Visitor<R> v) { return v.on(this); }
    }

    public static class Not implements Formula{
        public final Formula f;
        public Not(Formula f) { this.f = f; }
        public <R> R accept(Visitor<R> v) { return v.on(this); }
    }

    public static class And implements Formula{
        public final Formula left, right;
        public And(Formula left, Formula right) { this.left = left; this.right = right; }
        public <R> R accept(Visitor<R> v) { return v.on(this); }
    }

    public static class Or implements Formula{
        public final Formula left, right;
        public Or(Formula left, Formula right) { this.left = left; this.right = right; }
        public <R> R accept(Visitor<R> v) { return v.on(this); }
    }


    /**
     * @return set of variables used in a formula
     */
    // maintaining mutable state in a visitor
    public static Set<String> varSet (Formula f) {
```

just mutate it
— even shorter

```
        class VarSet implements Formula.Visitor<Void> {
            public final Set<String> vars = new HashSet<String>();
            public Void on(Var v) { vars.add(v.name); return null; }
            public Void on(Not n) { n.f.accept(this); return null; }
            public Void on(And a) { a.left.accept(this); a.right.accept(this); return null; }
            public Void on(Or o)  { o.left.accept(this); o.right.accept(this); return null;
}
        }
        VarSet vs = new VarSet();
        f.accept(vs);
        return vs.vars;
    }


    /**
     * @return set of variables used in a formula
     */
    // varSet #2: using an anonymous inner class and a final local variable
    public static Set<String> varSet2 (Formula f) {
        final Set<String> vars = new HashSet<String>();
        f.accept(new Formula.Visitor<Void>() {
            public Void on(Var v) { vars.add(v.name); return null; }
            public Void on(Not n) { n.f.accept(this); return null; }
            public Void on(And a) { a.left.accept(this); a.right.accept(this); return null; }
            public Void on(Or o)  { o.left.accept(this); o.right.accept(this); return null;
}
        });
        return vars;
    }
}
```

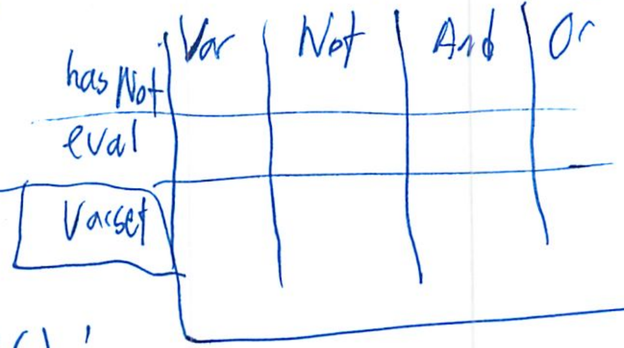
6.005 Recitation

10/6

design patters - ways of doing things

- ~~intern~~
- interpreter - 1st
- Visitor - 2nd

- two different ways of building an abstract syntax tree



~~class~~ Intepeter

```
interface Formula {
    boolean hasNot();
    boolean eval(Env e);
    Set<Var> varSet;
}

Class Var implements Formula {
    ~~hasNot~~ boolean hasNot () {
    }
    boolean eval (Env e) {
    }
}
```

②

```
Set <Var> varSet() {

}
```

- put all the var code together

Visitor pattern - easy to add type
- hard to add functions

- more like switch statement

- more like each row
  - put all hasNots together
- harder to add types

Wrong way to implement visitor pattern

~~~~
boolean hasNot(Formula f) {
  if (f instanceOf (Var)) {
    return false
  } else if (f instance of (Not)) {
    return true
  }
~~~~

Bad avoids static typing

(B)

Overloading - 2+ methods w/ same name but take diff input types

Interpreter
~~Overwriting~~ -
Overriding - defined in superclass or interface
- now in subclass have same signature
- give new/replace functionality

Visitor done correctly
- Create class hasNot - not a method
- ~~each~~ method for each type

```
class HasNot implements FormulaVisitor<Boolean>
    public Boolean onVar(Var v) {
        :
    }
    public Boolean onAnd(And a) {      <- did not have to downcast
        return hasNot(a.getLeft) || hasNot(a.getRight());
    }
    :
    public static hasNot(Formula f) {
        returns f.accept(new HasNot());   <- need constructor
    }
}
```

(4) Just had an interface for

```
                                  ↓placeholder
public interface FormulaVisitor <R> {
    public R    onVar (Var v);
     "      "   onNot (Not n);
                onAnd (And a);
                onOr  (Or o);
}

public interface Formula {
    public <R> R accept (FormulaVisitor <R> fv);
}
// implement in each class
public class Or implements Formula {
    public <R> R accept (FV <R> fv) {
        return fv.onOr(this);
    }
}
```

(Lots of boilerplate - need to review on my own)

Rep Invarient Reading

10/8

WP: Rep invarient: AAA Class Invarient

- invarient to constrain objects of class
- Methods of class should presere invarient
- Constrains state stored
- established during Construction
- all instances inherit class invarient

TA: Read Lecture 6 for more

Ohhh independent of underlying representation
(I just didn't put 2 + 2 together)

Read L6 for all

6.005 Elements of Software Construction | Fall 2011
Problem Set 4: Building a Sudoku Solver with SAT
Due: Tuesday, October 11 2011, 11:59 PM

The purpose of this problem set is to give you practice coding in Java, and to introduce you to immutable datatypes and SAT solving. This particular problem set is hard, so start early.

Do not change the signatures, specifications, or reps of any methods, classes, or packages that we have provided you, and do not add new public methods to the classes we have provided. Your code will be tested automatically, and will break our testing suite if you do so.

To get started, pull out the problem set code from SVN Admin.

Background

A *Sudoku puzzle* is a kind of Latin square. The aim is to complete a 9x9 grid with a digit between 1 and 9 in each square, so that -- as in a Latin square -- each digit occurs exactly once in each row and in each column. In addition, the grid is divided into 9 blocks, each 3x3, and each digit must also occur exactly once in each block. Sudoku is normally solved by reasoning, determining one step at a time how to complete an additional square until the entire puzzle is finished. Solving Sudoku by SAT is not very appealing for human players but works well on a computer.

A *propositional formula* is a logical formula formed from boolean variables and the boolean operators and, or and not. The satisfiability problem is to find an assignment of truth values to the variables that makes the formula true.

review 6.042

A *SAT solver* is a program that solves the satisfiability problem: given a formula, it either returns an assignment that makes it true, or says that no such assignment exists. SAT solvers typically use a restricted form of propositional formula called CNF.

A formula in *conjunctive normal form (CNF)* consists of a set of clauses, each of which is a set of literals. A literal is a variable or its negation. Each clause is interpreted as the disjunction of its literals, and the formula as a whole is interpreted as the conjunction of the clauses. So an empty clause represents false, and a problem containing an empty clause is unsatisfiable. But an empty problem (containing no clauses) represents true, and is trivially satisfiable. Complex

some way of expressing

Davis-Putnam-Logemann-Loveland (DPLL) is a simple and effective algorithm for a SAT solver. The basic idea is just *backtracking search*: pick a variable, try setting it to true, obtaining a new problem, and recursively try to solve that problem; if you fail, try setting the variable to false and recursively solving from there. DPLL adds a powerful but simple optimization called *unit propagation*: if a clause contains just one literal, then you can set the literal's variable to the value that will make that literal true. (There's actually another optimization included in the original algorithm for 'pure literals', but it's not necessary and doesn't seem to improve performance in practice.)

Wikipedia articles cover these topics nicely: Sudoku, CNF, DPLL, backtracking search, unit propagation.

in lecture notes described better

Overview

The theme of this problem set is to solve a sudoku puzzle. To do this, we'll:

1. Read in a text file with an incomplete sudoku puzzle, and represent the puzzle using an immutable abstract datatype.
2. Traslate the puzzle into a propositional formula, represented using immutable list data structures.
3. Solve the formula with the SAT problem solver.
4. Translate the formula back into a solution to the Sudoku puzzle.

Oh boy!

We're also providing you with packages that include implementations of immutable list data types and some of the propositional formula data types, as well as skeleton implementations of the Formula and Sudoku datatypes, which are in the sat.formula and sudoku package respectively. You only need to fill in the skeleton code to complete this problem set.

Your program should be efficient and should solve the 9x9 puzzles within 5 minutes (when assertions are turned off).

Note: You are **NOT** allowed to use an existing open source SAT solver as a part of your implementation in this problem set.

darn

Problem 1: Loading Sudoku Puzzles

Sudoku is an immutable datatype representing a Sudoku puzzle, with creator methods and observer methods. We have given you the specification of its methods, and its rep. It's your job to determine the rep invariant and implement the methods.

The datatype also has a factory method for loading a puzzle from a file. The file format is one line for each row of the puzzle, consisting of a sequence of digits (for known squares) and periods (for squares to be filled). We're providing you with two sample puzzles in this format, which are included in your repository along with this assignment.

> Important: You will use assertions in this assignment to check rep invariants (see Programming with Assertions). Assertions are turned

off by default in Java, which means that assert statements are completely ignored. To make sure assertions are on for all your JUnit tests:

1. Go to Preferences (either Windows / Preferences or Eclipse / Preferences).
2. Go to Java / JUnit.
3. Turn on "Add -ea to VM arguments when creating a new JUnit launch configuration."

If assertions don't seem to be enabled for a JUnit test, then enable them manually:

1. Go to Run / Run Configurations...
2. Find your JUnit class on the left side of the dialog box, and click on it.
3. Go to the Arguments tab, and enter `-ea` in the VM Arguments textbox.

a. [5 points] Write the rep invariant for Sudoku in a comment just after the instance fields, and implement the checkRep() method so that it checks your rep invariant using assert statements. (You can find examples of rep invariant comments and checkRep() in other classes in the provided code, such as Formula and Clause.)

c. [5 points]Write test cases for the two Sudoku constructors in SudokuTest, and then implement the constructors in Sudoku. Call checkRep() in your constructors to make sure the object you constructed satisfies the rep invariant, and implement Sudoku.toString().

d. [5 points]Write test cases for Sudoku.fromFile() and write the implementation of fromFile().

Problem 2: Representing SAT Formulas

Because SAT solving involves a lot of searching through different possible solutions, it turns out to be both more convenient and more memory efficient to implement using immutable data structures -- lists and maps that have no mutator methods. Instead of altering them (for example, adding an element to a list), you return a new object that has the modification, and that often shares much of its structure with the old list or map. In this pset, lists and maps are immutable, and you should NOT use the built-in List or Map classes from the Java API.

In this problem, you will use the immutable lists we provided you to implement formulas in conjunctive normal form.

a. [2 points] Write the datatype expression for Formula in a comment at the top of the Formula class. It should mention Formula, Clause, Literal, PosLiteral, and NegLiteral.

a. [3 points] Write test cases for Formula.

b. [10 points] Implement Formula.

Problem 3: SAT Solving

A SAT solver takes a propositional formula and finds an assignment to its variables that makes the formula true. In this problem, you will implement a SAT solver. You should develop and test your SAT solver independently of the Sudoku problem, i.e., don't feed it formulas produced from Sudoku puzzles (from Problem 4), but choose formula test cases appropriately.

a. [5 points] Write test cases for SATSolver.solve(). Some boolean formulas possible are:

- $(a \lor \neg b) \land (a \lor b)$ should return a: True, b: anything
- $(a \land b) \land (a \land \neg b)$ should return: null
- $(a \land b) \land (\neg b \lor c)$ should return: a: True, b: True, c: True.

This should get you started, but we expect to see more tests than the above.

b. [25 points] Implement solve(). Here is the pseudocode.

- If there are no clauses, the formula is trivially satisfiable.
- If there is an empty clause, the clause list is unsatisfiable -- fail and backtrack.
- Otherwise, find the smallest clause (by number of literals).
  - If the clause has only one literal, bind its variable in the environment so that the clause is satisfied, substitute for the variable in all the other clauses (using the suggested substitute() method), and recursively call solve().
  - Otherwise, pick an arbitrary literal from this small clause:
    - First try setting the literal to TRUE, substitute for it in all the clauses, then solve() recursively.
    - If that fails, then try setting the literal to FALSE, substitute, and solve() recursively.

Problem 4: Converting Sudoku to SAT

In order to use the SAT solver to solve a Sudoku puzzle, you need to represent the Sudoku puzzle as a propositional formula, using the Formula datatype you've already created. The variables in the formula are the occupies[i][j][k] variables in Sudoku's rep, which we'll write below as v_{ijk}. When the variable v_{ijk} is true, it means that square[i][j]==k in the final solution of the Sudoku grid.

You want a formula that will require all the following statements to be true, so you need to convert each of these statements into a formula and then AND them together.

- **Solution must be consistent with the starting grid.** You have some known entries in the square[][] array; each of those produces a clause in the formula. If square[0,2] starts out with digit 3, then you would have a clause containing the single positive literal v_{023}, which would require the SAT solver to set it true in the final solution. Don't create clauses for blank cells, so that the SAT solver is free to assign them.
- **At most one digit per square.** Without this requirement, you could get an assignment that sets both v_{000} and v_{001} to true, which would mean that cell (0,0) is occupied by two different digits at the same time. To prevent it, focus on one cell (i,j), look at each possible pair of digits k and k', and add to your formula a clause that guarantees that they can't both be in that square: $\neg v_{ijk} \vee \neg v_{ijk'}$.
- **In each row, each digit must appear exactly once.** For example, let's consider a 4x4 Sudoku grid, and let's focus on row i and digit k. Then the clause $v_{i0k} \vee v_{i1k} \vee v_{i2k} \vee v_{i3k}$ will guarantee that digit k appears *at least once* in row i. To guarantee that it appears *at most* once, we look at every pair of cells in the row, (i,j) and (i,j'), and require that they not both contain k: $\neg v_{ijk} \vee \neg v_{ij'k}$.
- **In each column, each digit must appear exactly once.** Like rows, but fixes the column j and the digit k. Generate one clause that guarantees k appears at least once in the cells of the column, and then one clause for each pair of cells in the column that guarantees they don't both contain k.
- **In each block, each digit must appear exactly once.** Same pattern as rows and columns, but the row and column indexes must vary over the cells within a given block.

requirements

Finally, after the SAT solver finds a satisfying assignment of true/false values to the v_{ijk} variables, you will need to convert this assignment back to numbers in a Sudoku grid. For every v_{ijk} that is assigned to true, you should have square[i][j]==k in the final grid.

a. [25 points] Implement Sudoku.getProblem(). Note: we don't require you to put tests for these two methods in SudokuTest, but you may if you want.

b. [5 points] Implement Sudoku.interpretSolution().

Problem 5: Putting it all together

[10 points] Test your code using the sample Sudoku puzzles (and any you want to add) through the Main method, and find out how long it takes your code to solve the puzzle. It should not take more than 5 minutes.

On what pc?

Extra

— like 6.034 games!

Make your Sudoku solver faster! (But without changing any method signatures.) This part will not affect your grade.

Before You're Done...

α, β ???

Double check that you didn't change the signatures of any of the code we gave you.

Make sure the code you implemented doesn't print anything to System.out. It's a helpful debugging feature, but writing output is a definite side effect of methods, and none of the methods we gave you to write should produce any side effects.

Make sure you don't have any outdated comments in the code you turn in. In particular, get rid of blocks of code that you may have commented out when doing the pset, and get rid of any TODO comments that are no longer TODOs.

Make sure your code compiles, and all the methods you've implemented pass all the tests that you added.

Does your code compile without warnings? In particular, you should have no unused variables, and no unneeded imports.

Make sure you check the last version of your code in SVN is the version you want to be graded.

Hints

- This problem set is quite challenging. You'll probably not succeed if you just start hacking and hope to make your way through it by brute force. Start early.
- Build your program incrementally. Try a very small SAT problem first that is small enough that you can trace the behavior of your solver with print statements if necessary. Try an unpopulated Sudoku puzzle before you try one that is partially completed, and try smaller puzzles (4x4) before you try the full-sized puzzle (9x9).
- You can create a standard propositional formula and convert it to CNF, but you'll find it easier if you generate CNF directly from the Sudoku grid, as described above.
- The simplest way to encode the puzzle in logic is to create one propositional variable for the possibility that each symbol can be in each square. So for a 9x9 puzzle, there will be 9x9x9 variables. This makes it clear that backtracking search alone will likely be completely infeasible, since the number of leaves of the search tree is 2 to the power of the number of variables. α, β
- To solve the full-sized Sudoku problem you may need to increase the amount of memory that the Java interpreter has allocated for heap space. Recall that every time you run your project (as a Java program, with JUnit, etc.) Eclipse creates a *run configuration* that specifies what to run and how to run it. To increase the maximum heap space for a run configuration, open the Run dialog (**Run → Run Configurations...**), select the relevant configuration, select the **Arguments** tab, and enter under VM arguments `-Xmx512m` (for example, which sets the maximum heap size to 512MB).

you: it's important to realize that using the strategy we gave for applying the axioms involves essentially the same effort it would take to construct truth tables, and there is no guarantee that applying the axioms will generally be any easier than using truth tables.

3.5 The SAT Problem

Determining whether or not a more complicated proposition is satisfiable is not so easy. How about this one?

$$(P \text{ OR } Q \text{ OR } R) \text{ AND } (\overline{P} \text{ OR } \overline{Q}) \text{ AND } (\overline{P} \text{ OR } \overline{R}) \text{ AND } (\overline{R} \text{ OR } \overline{Q})$$

The general problem of deciding whether a proposition is satisfiable is called *SAT*. One approach to SAT is to construct a truth table and check whether or not a **T** ever appears, but as for validity, this approach quickly bogs down for formulas with many variables because because truth tables grow exponentially with the number of variables.

Is there a more efficient solution to SAT? In particular, is there some, presumably very ingenious, procedure that determines in a number of steps that grows *polynomially*—like n^2 or n^{14}—instead of exponentially, whether any given proposition is satisfiable or not? No one knows. And an awful lot hangs on the answer. It turns out that an efficient solution to SAT would immediately imply efficient solutions to many, many other important problems involving packing, scheduling, routing, and circuit verification, among other things. This would be wonderful, but there would also be worldwide chaos. Decrypting coded messages would also become an easy task, so online financial transactions would be insecure and secret communications could be read by everyone. Why this would happen is explained in Section 8.9.

Of course, the situation is the same for validity checking, since you can check for validity by checking for satisfiability of negated formula. This also explains why the simplification of formulas mentioned in Section 3.2 would be hard—validity testing is a special case of determining if a formula simplifies to **T**.

Recently there has been exciting progress on *SAT-solvers* for practical applications like digital circuit verification. These programs find satisfying assignments with amazing efficiency even for formulas with millions of variables. Unfortunately, it's hard to predict which kind of formulas are amenable to SAT-solver methods, and for formulas that are *un*satisfiable, SAT-solvers generally get nowhere.

So no one has a good idea how to solve SAT in polynomial time, or how to prove that it can't be done—researchers are completely stuck. The problem of determining whether or not SAT has a polynomial time solution is known as the

"**P** vs. **NP**" problem.[1] It is the outstanding unanswered question in theoretical computer science. It is also one of the seven Millenium Problems: the Clay Institute will award you $1,000,000 if you solve the **P** vs. **NP** problem.

3.6 Predicate Formulas

3.6.1 Quantifiers

The "for all" notation, ∀, already made an early appearance in Section 1.1. For example, the predicate

$$\text{“}x^2 \geq 0\text{”}$$

is always true when x is a real number. That is,

$$\forall x \in \mathbb{R}.\ x^2 \geq 0$$

is a true statement. On the other hand, the predicate

$$\text{“}5x^2 - 7 = 0\text{”}$$

is only sometimes true; specifically, when $x = \pm\sqrt{7/5}$. There is a "there exists" notation, ∃, to indicate that a predicate is true for at least one, but not necessarily all objects. So

$$\exists x \in \mathbb{R}.\ 5x^2 - 7 = 0$$

is true, while

$$\forall x \in \mathbb{R}.\ 5x^2 - 7 = 0$$

is not true.

There are several ways to express the notions of "always true" and "sometimes true" in English. The table below gives some general formats on the left and specific examples using those formats on the right. You can expect to see such phrases hundreds of times in mathematical writing!

[1] **P** stands for problems whose instances can be solved in time that grows polynomially with the size of the instance. **NP** stands for nondeterministic polynomial time, but we'll leave an explanation of what that is to texts on the theory of computational complexity.

6.005 PS4

So rep invarient is things that must always be true about how we are representing data here

Going to define inline

Write test cases for constructors

This seems really silly

- well I guess can check Rep and toString...

At least they give us specs here

How do line break?

Got it to work

Need to test blank

And rep invariant

Ok so of [size] is actually size-1 ??

Is it

Should be easier to print arrays

- toString()?
  - does not help

So lenght 3 is 0 → 2 ?

②

Why does it let i == size?

i < size should not go there!

Confused i and j

So lenght = 4 means 0, 1, 2, 3

So fix check Rep

Sudoku values 1 -> 9

I don't get occupied

(A lot of work for something so simple...!)

Even testing constructors!

(Building my coding style...)

Now from file

File opening...

Read in char by char

Oh string equals

✓ works

Part 2 SAT Formula Representation

?So formula contains empty Formula?

So Formula
- made up of clauses
- which are made up of Literals

So each data type seperate!
- wow

How do I want to represent internally
- kinda my decision right

And then should be immutable

ImList of clauses
↑ here we go
written above

Must define empty or not

Recursive lists

(I think I am finally really getting this Java thing

(4)

Now how ~~the~~ to do add( )

- return new?
- but that is in underlying Clause type
- How about in Formula type?

I think just Return

- simple 1 line fns

But why is that private??

Or use old and call add.

So Piazza @412 has this

- can add private constructor

Ok keep moving

Iterator

- have not made before
- can just return one under it?

Need datatype expression + test cases

- how to test?
- quite simple...

Don't know which format...

(5)

Why are my tests not running?
Needed to restart Eclipse...

Oh AND or is in there

That is what datatype is for...

So built constructor test...
~~[illegible]~~ well built one...
So what is and (Formula p)
L conjunction of this and p
So build new ~~clause~~ formula

((this) AND (p))
?clause??

~~[illegible]~~ Is it CNF?
So just add p clause to this and return?
But here p is a formula
So append the lists...
Make new ImList and merge them

(6)

Can we add a IM List merge?

Hard since linked list format kinda

Need to think about more

element rest

How to take rest and chain to old

Would violate our proposal

- can append to start!

But still can't really append

Need to recopy clauses of one...

Or do we want new formula w/ params formula

- not allowed

Could slowly iterate over add clause...?

Jwang says to do that...

~~But~~ and - so no merge

And check that not duplicate

7

Or | is much harder

Need to distribute for each clause FOIL

? So actually need to go into clause

For each my literals
    For each their literals
        AND

Clauses are AND always, right? CNF?

Cool built Or w/o much trouble

Should have a better way to test Formula toString?

Changed it

That relies on printing order too

But easier?

We never built a parser to build CNF

But I guess not needed here

I'm not doing much destructive/fail/edge case testing.

(8)

NOT

Negation of whole thing

Mr. De Morgan

~~Why do we have to deal.~~

Oh converting will take you out of CNF

Need to convert back

How?

Do eaiser test cases

Not (A) =

Not (A V B) → Not A ∧ Not B

→ (Not A) ∧ (Not B)

grrr how to do?

do this first

How to ignore middle state

For each clause, for each literal add new Not clause

✓ Got it to work for NOT (A V B)

(7)

Now the example they give

Not[(A v B) ∧ (C)]

? So multiple terms

Need to conjugate with

Actually do we actually want intermediate form?
Need to OR between clauses
Do save temp thing with adding OR
(starting to like Java....)
Oh wow except for my stupid mistake did not
screw up basic case
Ok so its ~~[scribbled]~~ C ∧ (B V A)
which I did not expect....
How do do that
Add it to ∞ versions of ... but then has some size
limit
Take all previous and or before that

(10)

Hmm, does not look right either...

So if

$NOT((A \vee B) \wedge (C \vee D))$

is

$(\bar{A} \wedge \bar{B}) \vee (\bar{C} \wedge \bar{D})$ ← intermediate step

Like above

$(\bar{A} \vee \bar{C}) \wedge (\bar{A} \vee \bar{D}) \wedge (\bar{B} \vee \bar{C}) \wedge (\bar{B} \vee \bar{D})$

If we had it in ~~and~~ and AND / OR form would be easy!

If we do something like OR

$\bar{A} \vee \bar{C}$

with 4 level recurring...

But ours can be any level of stuff hmm...

11

Try the intermediate form thing I guess

Or what I was doing

Let me go through and one slowly

$$\frac{\overline{A}}{B} \vee \frac{D}{\overline{C}} \rightarrow \frac{\overline{B} \vee \overline{D}}{\overline{B} \vee \overline{C}}$$

Now when add a not on those

Only on new ones

; So multi dimensional!

Old and new lists

So that works only when I set temp Old.size fixed

- its does not mutate otherwise!

Its some interaction going on.

Clear better

Works on all 3 test cases!

getSize()

Should (in theory) be easier

Oh just return # of items in list!

Part 3 SAT Solver 10/9

I hate writing tests 1st → don't know proper format

Hmm they didn't use special literal classes

- we are returning Env
- So then empty is False
- Or just blank?

How does env look like?

└ can we set?

Need to build env compare method? or ToString?

↑✓ good

✓ Empty case in SAT Solver

Now I can write more test cases - before writing code!

And reg test cases in previous

All worked 1st try!

(13)

This is DPLL - pick smallest (ie single) clause

bind its values

Substitue using substitue

Recursivly call

~~Pick random~~

Or pick arbitrary + backtrack

i How, do we have a queue?

Just try building it

What does no empty clasuse is

No clause it empty clause inside

If empty itself - we are good

Im being very literal in how I am building my solver

May not be fastest / eaisest

This is quickly growing complicated

esp that did all psudocode at once

Also the ImList / Backtracking annoying to think about

(14)

Solve and substitute want IntList - not a formula!

- Why? Could you do it the other way?

So need to copy all clauses out of iterator?
No can call getClauses() on Formula

Now also need to build substitute

So run through and set to true and remove?

Yeah remove it if true

Oh built in Clause.reduce function

Must I rebuild clause - don't think I have to...

No, it says new list

Yeah can't just overwrite clause?

It seems I can - come back later

Move most code to other solve

Oh that *should* (not) work

Need to go through and make sure its all implemented

(15)

Stack overflow

- I don't think its ever returning

Start printing

Substituting not doing anything

Right since I did it wrong

- did not update, but did not error

Think might be working now...

- ~~Needs~~ Sub works

But env not being updated

Its setting false to true...

Flipped - works on case 3!

Now need to figure out how to test...

Is order of vars always same?

- No

Grrr - need to build equivilance checker?

Oh order does not matter

- grrr hard to do - esp quick + dirty

(16)

2 fails
- should return null
- no back tracking

1 looks to be getting wrong ans!

So substitute is wrong
(b, a) is going to null
but that is reduce?

No remember clause is a or b
So $a \vee \neg b$ returns true

Is working
Or substitute should do all clauses
else if contains not value? - ignore since is or??
Or set to false?
- remove it from there
So if checking for A if list A, B C — return
$\overline{A}$ if list A B C
?? ignore

(17)

Need to do any env

If see wrong on there, remove? or set to any?

Or perhaps do #2 back track 1st - or do others.

Backtrack - return null?

I think that is it - should test more w/ multilevel

(✓) #2

Now need to fix 1

So is True, True correct?

So I think its just a correct value.

(✓ #1)

So test some

Now need to do complex w/ backtracking

Hmm how to write test cases?

Look in notes

Socraties - human mortal. Ah this failed - look why + fix

(18)

$(\bar{S}H)(\bar{H}M)(S)(\bar{M})$

So $\bar{M}$

~~[illegible]~~ null $(\bar{H})(S)(H\bar{S})$

So $\bar{H}$

$(\bar{S})$ null (S)

So S

null $()$ ← isn't that failure?

Backtrack

(it jumps all the way up I think)

But is this supposed to be ~~[illegible]~~ null?

Oh data type is given for formula

Now should our example be true?

"All paths fail so theorm is valid"

that means no SAT???

IDK say null

But still don't know if backtracking works ...

(19)

I guess it does
- if 1 item is null - fail

If when we picked it did not work then it tries
Other one

Just need a better test case
To test switching
- where it does not work true
- but works false

$(A) \wedge (\neg B)$ - just assigns

$(A \vee C) \wedge (A \vee \neg C)$ - have

$(A \vee B) \wedge B \wedge C$

Say want A = False B = True C = True
- hard since DEF is clever!

~~(A ∧ B)~~ $(A \vee B) \wedge$ ~~...~~ $(B \vee C)$
Is C, True, B true, A ??

Emailed in

$A \wedge B \wedge \neg C$
? No simple ones it just solves

(20)

Instructor marked as good question...

$(\neg c, a)$ $(\sim c, \sim a)$ (b, c)

↑ Oh that might be it

Also recommends EclEmma plugin

So that did switch

Now proper values are null?

- yeah I think so

but all F should work

✓ ✓ B = true

C = false a = false

So it should Not fail - let me review

remove not working...

So I think I am removing wrong - not supposed to?

But no in ~~[illegible]~~ notes they show removal

Remove works fine as a black box

(21)

C true will cause it to fail

But it is back tracking <u>too much</u>

Should go back to C assignment

Need to do for other one

Ok combine single + Random literal

- never reason too keep seperate

Ok looks like no change...

Also it looks like substitue works

Is backtracking clearing aw right?

- no was printing wrong thing

So switching is getting wrong clauses

- need to do <u>new</u>

- also env is wrong

Now it overflows

- opps

Somehow variables are getting overwritten

- are not local

Then am not making copies properly

(2)

So need to somehow change pointer as well
(I thought local vars covered this ...)
It's not substituting not c correctly?
Not is identified correctly
Now remove not working?
- not removing nots properly
So I had gone wrong way w/ code
Think it works now
c = False
b = True
Yes a can be any ✓ Backtracking
So what did I do wrong originally?
- Did not check null on l when clause size = 1
? does this matter?
- Needed to sub the negation
- I think this was error
- Remove print code now ...

(23)

Part 4 Converting Sudoku to SAT

(Also need to check Rep on earlier ones)
 └ put in Java's TODO system so see

Occupies [i][j][k]

when true = V_{ijk}
 square[i][j] == k in final solution

So if square [0,2] = 3 then V_{023} = true

don't create for blank cell

Only V_{ijk} for 1 k can be true

$$\neg V_{ijk} \vee \neg V_{ijk'}$$ any

Each digit once in row - So for row i
 $V_{i0k} \vee V_{i1k} \vee V_{i2k} \vee V_{i3k} \vee V_{i4k}$ for 4
 So i, k fixed throughout
Same for column
and smaller block

(24)

and convert that back into grid

So how do you build this?

With variables?

Oh no test cases needed!

So iterate for each digit and add test cases

Can't do XOR!

So do start cells first

add both to occupies and to formula

Or should occupies already be built? hmm

How are literals represented - the variables?

So this does dim up to 9 -> Ok - could put in,

Why does occupies take variables?

Errors getting it started!

Why are we running off array?

Should k be -1? - no add 1 to size then

0 is also blank

- should record earlier as -1

(25)

Cool reading starting grid ✓

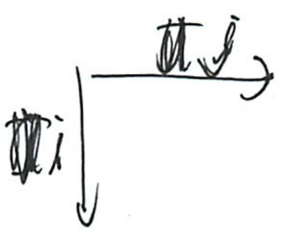
One 1 digit at a time! ✓

Row - fixed i ✓

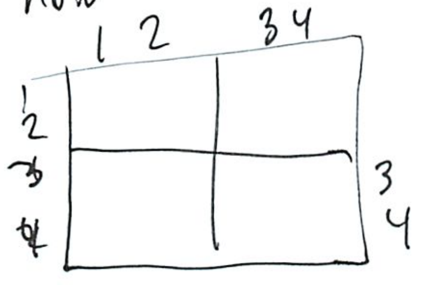
- just needed to think about a bit

Now column

V_{0jk} ← fixed



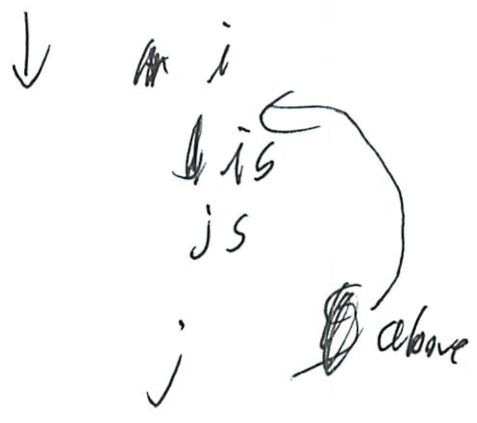
Block now



\# of blocks = dim
Size of blocks = dim

So for each block ↓
for each size ↓
go →

For ea
go → blocks
Start fresh :)



(26)

Need to build cell addressing

(dim - 1) + i is ?

?

j · dim

Now need for each k

So 3 in one of each of cells

So add for k

L but further up

Since want or's over that

Opps no -1 in j·dim

L since 0 indexed

✓ Nice block constraint :)

Occupies is useless - I build myself

- or did I do this wrong ??

Now need to interpret ...

what does it give back?

Oh an env

I think I got this - need to go 1 by 1 + parse

27

Darn bindings invisable

How else can I iterate through an env?

Make a method!

Ah someone asked @ ~~4~~ 421

TAs want us to iterate over occupies array

- Ok 2nd thought....

Ah this takes care of string matching when size > 9

Oh I did ks wrong all along...

ks index is row value

So k = 0 is invalid!

Why is going to k = 5?

Variable copy error

Some sols still blank!

~~Some~~ ~~still~~ still and others have been changed!

Fixed the + 1

Now getting a lot all 4s!

Atleast no replacing!

(28)

I think my loading is faulty...

How can multiple variables be true?

What does env return if not in there? — And not in list

No its in the true list

But clause is in there 134 v 133 v 132 v 131

Now back to SAT solver...

Or am I thinking about it wrong??

Remember clause = or statement

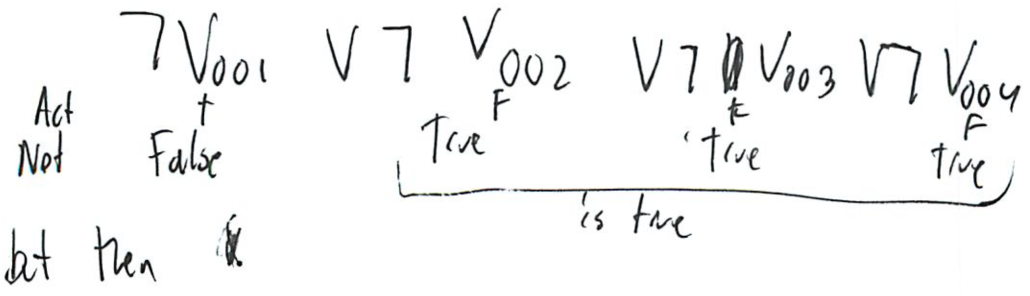
- So did at least once

Now need at most once

- did starting grid ✓
- at most one square

$\neg V_{ijk} \vee \neg V_{ijk'}$

So if was 001 true then test is



but then

(29)

It says each pair

$\neg V_{001} \vee \neg V_{002}$
true

$\neg V_{001} \vee \neg V_{003}$
true

$\neg V_{001} \vee \neg V_{004}$
true

So can't have both of them be true
or other would be false

More 6.004-style neg logic

Can keep the Or one since need one
But this is prob elsewhere
— but keep it anyways for now

(V) Made those

Now need at most for each row + column + block

So it looks good except for block - many duplicate...

Oh forgot the iff ≠

I look like I have an ans !!!

That looks good (V) Now test the others

(30)

Looks to be going really, really slow...

- Just the simple clauses take forever!

Substitute prob taking forever!

- try removing "remove"
- retest
- no screws it all up...

Emailed in

I'm prob doing something wrong on the lists

Go to office hrs + move on

Oh - sat only

Tue I'm busy

I think I should rewrite substitute

But I prefer in lab hrs

Jwang helped me at on Sub rewrite

- if size != 0 skip it as well

- still 1600 ms - don't know w/ all printing...
- removed some 394 ms - better!
- 339 ms no printing

(31)

Still need more ideas...

Too many clauses in

- already got rid of 1 set

Or inefficent copying/backtracking

I think I did it all correct...

Takes a long time just to make the ~~[illegible]~~ SAT formula

~20 sec

Still think its solver

Its the sub that is slow

Call sub start print end - and the hang is still in sub start

? Then I have too many params?

Too many clauses to copy ??

But I followed the instructions

Piazza 440 - put add to Grid last?

377ms

Is Jweng's reduce wrong?

Try rewriting w/ iterator

(32)

4 is now 182 ms w/ printing

Ah subbing is far faster

But mem error

- mem just goes to stratosphere

Going back to remove way slower

Moving to 1 GB memory it works!

So remove debugging + test

Though on all blank some # still blank

- add that requirement back

Though 6.034 - most constrained 1st

Now check to dos

I actually don't use occupies much...

Now run again w/o debug

6.005 Review
Lectures

10/12

- Project 1
  - Piano player from ABC music files
  - teams of 3
    - find people to sign up w/
  - 1st milestone due Tue - warm up coding
    - design
  - no lectures/recitations next week
  - TA meetings Wed, Thur
- Quiz this Fri 11-12 Walker 3rd Floor
- 1st return in deadline Sun night
  - to get before drop date

- [ ] methods
- [ ] mutable objs
- [ ] ADTs

How do you go about creating programs?

simple **Methods** → **Mutable Objs** → **ADT** complex

L1-3 | - state machine L4-5 | L6-8

②

Approach for all types

1. Think about problem
   - design
   - Write specs
     - including return types
     - exceptions can generate
       - or use special return values
       - checked exceptions - need catch, try
         - or caller of method can throw exception as well
       - unchecked exceptions - null pointer
         - array bounds
         - signs of programing flaws
   - preconditions
     - on parameters or state
   - post conditions
     - what implementer guarentees
   - modifies
     - framing condition
     - a post condition

(3)

2. Testing
- unit testing - test 1 method, not whole program
- input partitioning
  - finding ones w/ similar conditions
- boundry values
- Should test before coding - "black box"
  ↳ not written yet
  - at end will forget
  - still want to do for regression testing
- psy. benefit - just focus on getting all greens
- learn about what its like to be a client of you
- Sometimes hard to test

3. Coding - writing body of method
- Static checking - Eclipse does before you run
- arrays, lists, maps
- mutable + immutable strings

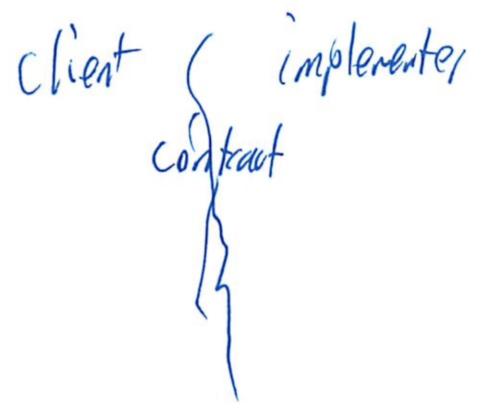
4. Retest - finish testing
trying to cover every line of code "code coverage"
"glass box coding"

(4)

~~Like~~

Like how you force backtracker depends on your solver

Don't care about env order



Tester not supposed to violate contract
 (implementer not bound to return anything

If add checked exception client must deal w/
 - no longer a pre-condition
 - now a post condition

In Java anything that ends in "Error" is a unchecked exception
 - also Null pointer, Array out of bound

Checked exception
 - happening outside
 - not bugs
 - predictable
 - IO Exception

- End of File Exception
- File Not Found exception

Mutable objects

- state machines
- grammars
  - guide your implementation of lang
- testing: test all states + all transitions + all paths

  all states ↑ too basic ⊗ ; all transitions ↑ more effective ✓ ; all paths ↑ infeasible ⊗

- enumerations
  - tokens in Lexer

ADTs

datatype definitions

RatNum = ______

↑ design "rep" - concrete classes that will represent it

could do int × int (↑ numerator, ↑ denominator)

String = char[]

or

= char[] × int × int (↑ start, ↑ end)

(6)

Usually want to build a constructor

= SimpleString(a: char[])

= Substring(a: char[], start: int, end: int)

Can write an interface

Then implement the interface

- when have >1 implementation

Operations

- creators
- producers
- mutators
- observers

2 levels

- spec level } operations at spec level
- rep level

Interfaces can be implemented by anyone

say t...

(7)

Interpreter / Visitor trade off

Varients are alt. representations
- can be ind - don't interact
- in formula do interact very much

Invarient property that is always true

```
int x = 5;
```

x will always be an int — but not always be 5

```
String s = ""
// s != null
```

s instance of String

Immutable

dim2 = lenght ← a requirement any you have places

can't change w/ final

and underlying object does not change

Chief threat to invariance/immutability = exposure to program
- underlying data is often mutable

6.005 Recitation
Test Review

10/13

- Specs

Lots of people had trouble on P-set

Think of audience

- its like you are using API

You don't care about underlying structure / implementation
- or who calls it (does not matter if a key pressed, Some one else can change it later)

Care about
- preconditions
- modifies
- returns
- params

You can comment ~~what~~ how implemented but do it line level

Could mention states in class or method

(2)

Checked/Unchecked exceptions

- question on practice test
- Checked - expect that this could happen
  - network goes down
  - file locked, no permissions
- unchecked
  - errors you don't think will ever happen
- checked do try () catch ()
- or throw up another level

Visitors

Why visitors? Big syntax mess!

	Var	hasNot	And	Or
eval				
hasNot				
same				

③

Interperter column-centric
all stuff about var, Not is same

Visitor row centric

all stuff for eval, hasNot, etc goes together

What do you think you will modify more?

Visitor + Interperter are both patterns
- use when want to perform operations on recursive data types

Recursive data type
does not need to be recursive
can be just abstract data type

Expr = Var(s: String)
+ Not(e: Expr)
+ And(l: Expr, r: Expr)

Expression can be defined by an expression
- an expression can be an AND - which is made up of 2 expressions

(4)

Implementation

Example on practice test

Look at arithmetic expressions
- simpler than PS3

Expr = Const(val : double)
+ Plus (left : Expr, right : Expr)

```
public Const (double val) {
    this.val = val;
}
```

~~public Const ( ) { }~~

we could do this but the way we defined Const **needs** a double

we could set default of 1.0

```
public Const () {
    this.val = 1.0;
}
```

(5)

Sometimes we want to expose constructor

- Sometimes just a method

This syntax is different from datatype production

- When we are writing a grammar
- like match parens example

Expr ::= LParen Expr RParen | Expr Expr

(| = or)

Look at lecture notes for visitor

- Interface for whole thing
- Then class for each

Interface Expr

public class Const implements Expr {

accept(ExprVisitor v) ...

" " Plus " "

" " Minus " "

interface ExprVisitor

visit (Const); ← method

visit (Plus);

visit

(6)

public class Eval Visitor implements Expr Visitor

visit (~~[illegible]~~ Const c) {

Quiz on website diff topics

More Java specific than we will be

Practice 10/13/2011

Massachusetts Institute of Technology
6.005: Elements of Software Construction
Fall 2010
Quiz
Wednesday, 17 November 2010

Name:____

Athena User Name: ______________________________

Instructions

longer

This quiz is 80 minutes long. It contains 31 questions in 13 pages (including this page) for a total of 100 points.

Please check your copy to make sure that it is complete before you start. Turn in all pages, together, when you finish. Write your name on the top of every page. Please write neatly. No credit will be given if we cannot read what you write. Good luck!

Question Name	Maximum Points	Question Numbers	Points Given
True/False	30	1-15	
Specifications	6	16	
Module Dependency	9	17-19	
Exceptions	9	20-22	
Immutability	6	25	
Visitor Pattern	15	26-28	

Coverage slightly different
less Java

Name: ____

True/False [30 points, 2 points each]

Circle the correct answer:

1. T / F
   The representation invariant should hold at all times during the execution of a method.

2. T / F
   A precondition can simplify the implementation of a method.

3. T / F
   Methods of immutable types always have empty modifies clauses in their specifications.

4. T / F
   A Java interface type cannot be used in a typecast since every object's type is a concrete class.

5. T / F
   The representation of an abstract data type (ADT) is exposed if one of its methods returns a mutable object.

6. T / F
   An immutable ADT can have a mutable representation.

7. T / F
   If you can prove that every constructor produces a well-formed object that satisfies the representation invariant (RI) and that every mutator preserves the RI of the argument object, then RI assertions will never fail.

8. T / F
   In Java, an interface can 'extend' another interface.

Name: ______________________________

9. T / F
Is it possible that the code of `A.foo` is executed during the evaluation of `(new B()).foo()` in the following?

```
class A {
        ...
     public foo() { ... }
}
class B extends A {
        ...
     public foo() { ... }
}
```

No override

For questions 12-15 assume
- `B` extends `A`
- `A` has a defined `apply()` method and so does `B`
- `a` refers to an object whose compile-time ~~type~~ and run-time type is `A`
- `b` refers to an object whose compile-time and run-time type is `B`.

10. T / F ✓
`b.apply(a)` can behave differently than `((B) b).apply(a)`.

11. T / F
`b.apply(a)` can behave differently than `((A) b).apply(a)`.

12. T / F ✓ Why not
`b.apply(a)` can behave differently than `b.apply((B) a)`.

13. T / F ✓
`b.apply(a)` can behave differently than `((B) b).apply((A) a)`.

I should study Interface Vs. Visitor

Prob a trick here

Overriding = Runtime type

Overloading = Compile/Declared type

Name: ______________________________

Specifications [6 points]

14. Write a precondition or requires clause for the method `removeDuplicates`, so all duplicates from `List lst` are removed. Your precondition should be non-trivial, e.g., it cannot be that the input `lst` is a list without duplicates. Feel free to write your specification in plain English.

T can have dupes

```
public static void removeDuplicates(List lst) {
      if (lst == null || lst.size() == 0) return;

      List copy = new ArrayList(lst);
      Iterator elements = copy.iterator();
      Object pre = elements.next();

      while(elements.hasNext()) {
            Object nex = elements.next();
            if (pre.equals(nex)) lst.remove(nex);
            else pre = nex;
      }
}
```

Requires clause or precondition:

↓ don't need but can have

lst = a List with > 0 elements and not null

& any thing else

Must be sorted ✓

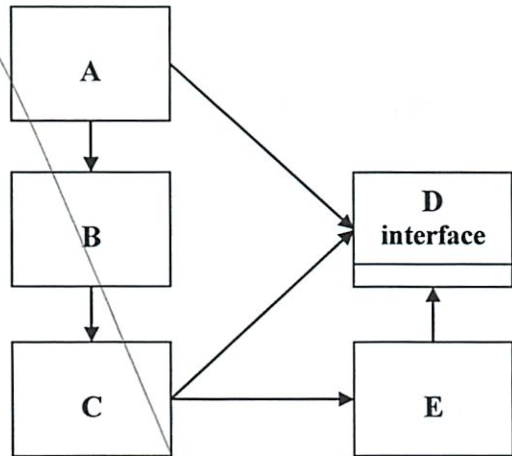
Re Duplicates

Name: ______________________________

Module Dependences [9 points]

Not on

Examine the Module Dependence Diagram (MDD) below and answer questions 17-19.



Arrow means "depends on." Which parts need to be examined and possibly changed (answer with `A`, `B`, `C`, `D`, and/or `E`) if

15. [3 points] Only the specification of `B.method1()` is changed?

16. [3 points] Only the implementation of `E.method1()` is changed?

17. [3 points] Interface `D` is replaced with a new part `F`?

Name: ______________________________

Exceptions [9 points]

For each snippet of code in questions 20-21, indicate whether the exception being thrown should be a checked exception or unchecked and explain briefly why.

18. [3 points]

```
if (!checkRepInv())
    throw new AException();
```

Unchecked – error – should never be reached ✓

19. [3 points]

```
if (fileNotFound())
   throw new BException();
```

Checked – expected ✓

20. [3 points]

```
// requires: x    >=   0
public void insert(int x) {
   if ( x < 0 ) throw new CException();
}
```

must be checked

but violating this is violating precondition which is a legal error ✓

They wanted this

Name: ______________________________

Immutability [6 points]

Ben Bitdiddle wants to make an immutable list which takes in a List and stores it in a final variable as shown in the code below:

```
public final class ImmutableList<T>{
     private final List<T> backingList;

     public ImmutableList(List<T> list){
         backingList = list;
     }
     public boolean equals(Object o) {
         return backingList.equals(o);
     }
     public int hashCode() {
         return backingList.hashCode();
     }
     public T get(int index) {
         return backingList.get(index);
     }
     public int size() {
         return backingList.size();
     }
}
```

Unfortunately, his Immutable list actually is mutable. Allysa P. Hacker suggests he change his constructor to:

```
     public ImmutableList(List<T> list){
         backingList = new ArrayList<T>(list);
     }
```

Copy

21. Is Ben's list now immutable? Explain why or why not. If it is not, suggest a fix to make the list immutable.

?. When it returns get is it a pointer or the value (forget how list implemented) doesn't really hurt to make a preemptive deep copy

✓ elements of a list

deep copy

Name: ______________________________

Visitor Pattern [15 points]

Ben is doing Lab 2 for 6.005. He is given an `Expr` class, and its subclasses `Const`, `Plus` and `Minus`:

```
public abstract class Expr {
     /* Empty Class */
}
public class Const extends Expr {

    private final double value;

    public Const(double value){
     this.value = value;
    }
    public double getValue() {
        return value;
    }
}
public class Plus extends Expr {

    private final Expr left, right;

    public Plus(Expr left, Expr right){
     this.left = left;
     this.right = right;
    }
    public Expr getLeft() {
        return left;
    }
    public Expr getRight() {
        return right;
    }
}
```

`Minus` has the exact same internal definitions as the one shown above for `Plus`.

Ben decides to implement the visitor pattern for expressions. He creates the following visitor interface:

```
public interface ExprVisitor<T>{
    public T visit(Const c);
    public T visit(Plus plus);
    public T visit(Minus minus);
    public T visit(Expr expr);
```

Name: ____________________

```
    }
```

Instead of implementing the `accept(...)` method in each of the subclasses of `Expr`, Ben decides to save time and implement the `accept(...)` method in the top-level `Expr` class. This way, all the subclasses of `Expr` don't need to have an `accept(...)` method. Here is his implementation:

very bad sign

abstract = interface?

```
public abstract class Expr {
    public <T> T accept(ExprVisitor<T> visitor) {
        return visitor.visit(this);
    }
}
```

22. [5 points] To get this method to compile, Ben has already added one extra `visit(...)` method to his `ExprVisitor` interface that would normally not be there. Which one is it, and why did he need to add it?

public T visit(Expr expr) ✓

since this is an intaface would not normally need it - but he added it

? Oh not really - type of this = Expr, need a visit(x)

23. [5 points] Ben argues that his implementation works, because the `visit(Expr)` method will never get called. He asserts that the accept method for `Plus` would call the visitor's `visit(Plus)` method automatically, that the accept method for `Minus` would call the visitor's `visit(Minus)` method automatically, and so on. Is Ben's reasoning correct? Explain.

No ✓ visit(Expr) cold be called always called

on an entire expression

- but what does it do

- we never talked abat

Java decides overloaded at combie time

Name: ______________________________

24. [5 points] Now, assume that Ben has fixed the issues (if any) in his implementation of the visitor pattern for expressions, and he wants to create an expression and apply a visitor on it. Also assume that Ben has already implemented `EvaluateExprVisitor`. See the code excerpt below:

```
Expr e = new Plus (new Const(1.0), new Const(2.0));
ExprVisitor<Double> evaluator = new
                                   EvaluateExprVisitor();

Double result = /* Your Code Here */
```

Complete the code here by using `evaluator` to set `result` to be equal to the computed value of the expression `e`.

~~evaluator.getValue()~~

~~evaluator.getLeft().getValue() +~~
~~evaluator.getRight().getValue()~~

e.accept(evaluator) ✓ knew it should be more simple
expr

Can't evaluator.visit(e)
is Expr

No type casting

Massachusetts Institute of Technology
6.005: Elements of Software Construction
Fall 2010
Quiz
Wednesday, 17 November 2010

Name:____

Athena User Name: ______________________________

Instructions

This quiz is 80 minutes long. It contains 31 questions in 13 pages (including this page) for a total of 100 points.

Please check your copy to make sure that it is complete before you start. Turn in all pages, together, when you finish. Write your name on the top of every page. Please write neatly. No credit will be given if we cannot read what you write. Good luck!

Question Name	Maximum Points	Question Numbers	Points Given
True/False	30	1-15	
Specifications	6	16	
Module Dependency	9	17-19	
Exceptions	9	20-22	
Immutability	6	25	
Visitor Pattern	15	26-28	

Name: ______________________________

True/False [30 points, 2 points each]

Circle the correct answer:

1. T / F
   The representation invariant should hold at all times during the execution of a method. *Need only hold at the end and beginning.*

2. T / F
   A precondition can simplify the implementation of a method.

3. T / F
   Methods of immutable types always have empty modifies clauses in their specifications. *Can modify arguments ('this' should not change).*

4. T / F
   A Java interface type cannot be used in a typecast since every object's type is a concrete class.

5. T / F
   The representation of an abstract data type (ADT) is exposed if one of its methods returns a mutable object. *Could copy before returning it.*

6. T / F
   An immutable ADT can have a mutable representation. *Just don't expose it!*

7. T / F
   If you can prove that every constructor produces a well-formed object that satisfies the representation invariant (RI) and that every mutator preserves the RI of the argument object, then RI assertions will never fail. *Rep exposure.*

8. T / F
   In Java, an interface can 'extend' another interface.

Name: ____________________

9. <u>T</u> / F
Is it possible that the code of `A.foo` is executed during the evaluation of `(new B()).foo()` in the following?

```
class A {
    ...
    public foo() { ... }
}
class B extends A {
    ...
    public foo() { ... }
}
```

Could call super.foo() in B.

For questions 12-15 assume

- `B` extends `A`
- `A` has a defined `apply()` method and so does `B`
- `a` refers to an object whose compile-time type and run-time type is `A`
- `b` refers to an object whose compile-time and run-time type is `B`.

10. T / <u>F</u>
`b.apply(a)` can behave differently than `((B) b).apply(a)`.

11. T / <u>F</u>
`b.apply(a)` can behave differently than `((A) b).apply(a)`.

12. <u>T</u> / F
`b.apply(a)` can behave differently than `b.apply((B) a)`.

13. T / <u>F</u>
`b.apply(a)` can behave differently than `((B) b).apply((A) a)`.

MANTRA: Overriding == Runtime Type, Overloading == Compile/Declared Type

Name: ____________________

Specifications [6 points]

14. Write a precondition or requires clause for the method `removeDuplicates`, so all duplicates from `List lst` are removed. Your precondition should be non-trivial, e.g., it cannot be that the input `lst` is a list without duplicates. Feel free to write your specification in plain English.

```
public static void removeDuplicates(List lst) {
    if (lst == null || lst.size() == 0) return;

    List copy = new ArrayList(lst);
    Iterator elements = copy.iterator();
    Object pre = elements.next();

    while(elements.hasNext()) {
        Object nex = elements.next();
        if (pre.equals(nex)) lst.remove(nex);
        else pre = nex;
    }
}
```

Requires clause or precondition:

Requires input to be sorted

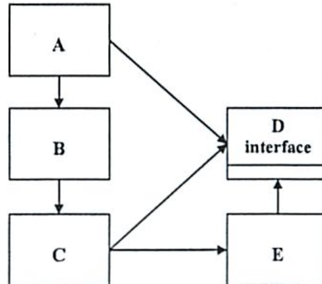
OR

Requires duplicates to be adjacent

Name: ______________________

Module Dependences [9 points]

Examine the Module Dependence Diagram (MDD) below and answer questions 17-19.



Arrow means "depends on." Which parts need to be examined and possibly changed (answer with A, B, C, D, and/or E) if

15. [3 points] Only the specification of `B.method1()` is changed?

    A (B)

16. [3 points] Only the implementation of `E.method1()` is changed?

    None (E)

17. [3 points] Interface D is replaced with a new part F?

    A, B, C, E

Name: ______________________

Exceptions [9 points]

For each snippet of code in questions 20-21, indicate whether the exception being thrown should be a checked exception or unchecked and explain briefly why.

18. [3 points]

```
if (!checkRepInv())
    throw new AException();
```

Unchecked exception. Programmer cannot be expected to fix this.

19. [3 points]

```
if (fileNotFound())
   throw new BException();
```

Checked exception. Programmer should handle this exception.

20. [3 points]

```
// requires: x    >=   0
public void insert(int x) {
   if ( x < 0 ) throw new CException();
}
```

Unchecked exception. Programmer cannot be expected to fix this.

Name: ______________________

Immutability [6 points]

Ben Bitdiddle wants to make an immutable list which takes in a List and stores it in a final variable as shown in the code below:

```
public final class ImmutableList<T>{
     private final List<T> backingList;

     public ImmutableList(List<T> list){
         backingList = list;
     }
     public boolean equals(Object o) {
         return backingList.equals(o);
     }
     public int hashCode() {
         return backingList.hashCode();
     }
     public T get(int index) {
         return backingList.get(index);
     }
     public int size() {
         return backingList.size();
     }
}
```

Unfortunately, his Immutable list actually is mutable. Allysa P. Hacker suggests he change his constructor to:

```
     public ImmutableList(List<T> list){
         backingList = new ArrayList<T>(list);
     }
```

21. Is Ben's list now immutable? Explain why or why not. If it is not, suggest a fix to make the list immutable.

Ben's list is still mutable. `T`, *i.e., the elements of the list themselves could be mutable. We need to make a deep copy.*

Name: ______________________

Visitor Pattern [15 points]

Ben is doing Lab 2 for 6.005. He is given an `Expr` class, and its subclasses `Const`, `Plus` and `Minus`:

```
public abstract class Expr {
     /* Empty Class */
}
public class Const extends Expr {

    private final double value;

    public Const(double value){
     this.value = value;
    }
    public double getValue() {
        return value;
    }
}
public class Plus extends Expr {

    private final Expr left, right;

    public Plus(Expr left, Expr right){
     this.left = left;
     this.right = right;
    }
    public Expr getLeft() {
        return left;
    }
    public Expr getRight() {
        return right;
    }
}
```

`Minus` has the exact same internal definitions as the one shown above for `Plus`.

Ben decides to implement the visitor pattern for expressions. He creates the following visitor interface:

```
public interface ExprVisitor<T>{
    public T visit(Const c);
    public T visit(Plus plus);
    public T visit(Minus minus);
    public T visit(Expr expr);
}
```

Name: ______________________________

Instead of implementing the `accept(...)` method in each of the subclasses of `Expr`, Ben decides to save time and implement the `accept(...)` method in the top-level `Expr` class. This way, all the subclasses of `Expr` don't need to have an `accept(...)` method. Here is his implementation:

```
public abstract class Expr {
    public <T> T accept(ExprVisitor<T> visitor) {
        return visitor.visit(this);
    }
}
```

22. [5 points] To get this method to compile, Ben has already added one extra `visit(...)` method to his `ExprVisitor` interface that would normally not be there. Which one is it, and why did he need to add it?

    He added the `visit(Expr)` *method.*

    The declared type of '`this`' *in* `Expr` *is* `Expr`*; The Java compiler checks for type safety and requires that a* `visit(X)` *method be present in the visitor interface, where X is Expr or a super-type of Expr.*

23. [5 points] Ben argues that his implementation works, because the `visit(Expr)` method will never get called. He asserts that the accept method for `Plus` would call the visitor's `visit(Plus)` method automatically, that the accept method for `Minus` would call the visitor's `visit(Minus)` method automatically, and so on. Is Ben's reasoning correct? Explain.

    Ben is wrong.

    `visit(Expr)` *is always called. This is because Java decides which overloaded method to call at* <u>*compile-time*</u>*; it thus uses the declared type of the* arguments(s) *to decide which method gets called.* '`this`' *refers to an* `Expr` *declared type in* `Expr`*, so we always call* `visit(Expr)` *in an* `accept(...)` *method.*

Name: ______________________________

24. [5 points] Now, assume that Ben has fixed the issues (if any) in his implementation of the visitor pattern for expressions, and he wants to create an expression and apply a visitor on it. Also assume that Ben has already implemented `EvaluateExprVisitor`. See the code excerpt below:

```
Expr e = new Plus (new Const(1.0), new Const(2.0));
ExprVisitor<Double> evaluator = new
                                EvaluateExprVisitor();

Double result = /* Your Code Here */
```

Complete the code here by using `evaluator` to set `result` to be equal to the computed value of the expression `e`.

`e.accept(evaluator);`

`evaluator.visit(e)` fails because e is declared to be `Expr`.

`Evaluator.visit((Plus) e)` fails to use the elegance of double dispatch; typecasts in general should be/can be avoided with visitor patterns, and this typecast depends on `e` being a `Plus` instance.