

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 1: Static Checking

Hailstone Sequence

Computing Hailstones

Types

Static Typing

Static Checking, Dynamic Checking, No Checking

Surprise: Primitive Types Are Not True Numbers

Arrays and Collections

Iterating

Methods

Mutating Values vs. Reassigning Variables

Documenting Assumptions

Hacking vs. Engineering

The Goal of 6.005

Why we use Java in this course

Summary

Reading 1: Static Checking

Objectives for Today's Class

Today's class has two topics:

- static typing
- the big three properties of good software

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
 $2^n, 2^{n-1}, \dots, 4, 2, 1$ 
5, 16, 8, 4, 2, 1
7, 22, 11, 34, 17, 52, 6, 13, 40, ...? (where does this stop?)

```

Because of the odd-number rule, the sequence may bounce up and down before decreasing to 1. It's conjectured that all hailstones eventually fall to the ground – i.e., the hailstone sequence reaches 1 for all starting n – but that's still an open question (https://en.wikipedia.org/wiki/Collatz_conjecture). 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 some code for computing and printing the hailstone sequence for some starting n . We'll write Java and Python side by side for comparison:

```

// Java
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);

```

```

# Python
n = 3
while n != 1:
    print n
    if n % 2 == 0:
        n = n / 2
    else:
        n = 3 * n + 1
print 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.
- Java requires semicolons at the ends of statements. The extra punctuation can be a pain, but it also gives you more freedom in how you organize your code – you can split a statement into multiple lines for more readability.
- Java requires parentheses around the conditions of the `if` and `while`.
- Java requires curly braces around blocks, instead of indentation. You should always 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.

Types

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 $\pm 2^{31}$, or roughly ± 2 billion)
- `long` (for larger integers up to $\pm 2^{63}$)

- `boolean` (for true or false)
- `double` (for floating-point numbers, which represent a subset of the real numbers)
- `char` (for single characters like '`'A'`' and '`'$'`)

Java also has **object types**, for example:

- `String` represents a sequence of characters, like a Python string.
- `BigInteger` represents an integer of arbitrary size, so it acts like a Python integer.

By Java convention, primitive types are lowercase, while object types start with a capital letter.

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 no matter how they're written. Here are three different syntaxes for an operation in Python or Java:

- As an *infix, prefix, or postfix operator*. For example, `a + b` invokes the operation `+ : int × int → int`.
- As a *method of an object*. For example, `bigint1.add(bigint2)` calls the operation `add : BigInteger × BigInteger → BigInteger`.
- As a *function*. For example, `Math.sin(theta)` calls the operation `sin : double → double`. Here, `Math` is not an object. It's the class that contains the `sin` function.

Contrast Java's `str.length()` with Python's `len(str)`. It's the same operation in both languages – a function that takes a string and returns its length – but it just uses 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. Methods can also be overloaded. Most programming languages have some degree of overloading.

Static Typing

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 `int`s, 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 while you're still typing.

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

Static typing is a particular kind of **static checking**, which means checking for bugs at compile time. 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 typing 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"
```

that tries to multiply two strings, then static typing will catch this error while you're still programming, rather than waiting until the line is reached during execution.

Static Checking, Dynamic Checking, No Checking

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

- **Static checking**: the bug is found automatically before the program even runs.
- **Dynamic checking**: the bug is found automatically when the code is executed.

- **No checking** : the language doesn't help you find the error at all. You have to watch for it yourself, or end up with wrong answers.

Needless to say, catching a bug statically is better than catching it dynamically, and catching it dynamically is better than not catching it at all.

Here are some rules of thumb for what errors you can expect to be caught at each of these times.

Static checking can catch:

- syntax errors, like extra punctuation or spurious words. Even dynamically-typed languages like Python do this kind of static checking. If you have an indentation error in your Python program, you'll find out before the program starts running.
- wrong names, like `Math.sine(2)` . (The right name is `sin` .)
- wrong number of arguments, like `Math.sin(30, 20)` .
- wrong argument types, like `Math.sin("30")` .
- wrong return types, like `return "30";` from a function that's declared to return an `int` .

Dynamic checking can catch:

- illegal argument values. For example, the integer expression `x/y` is only erroneous when `y` is actually zero; otherwise it works. So in this expression, divide-by-zero is not a static error, but a dynamic error.
- unrepresentable return values, i.e., when the specific return value can't be represented in the type.
- out-of-range indexes, e.g., using a negative or too-large index on a string.
- calling a method on a `null` object reference (`null` is like Python `None`).

Static checking tends to be about types, errors that are independent of the specific value that a variable has. A type is a set of values. Static typing guarantees that a variable will have *some* value from that set, but we don't know until runtime exactly which value it has. So if the error would be caused only by certain values, like divide-by-zero or index-out-of-range then the compiler won't raise a static error about it.

Dynamic checking, by contrast, tends to be about errors caused by specific values.

Surprise: Primitive Types Are Not True Numbers

One trap in Java – and many other programming languages – is that its primitive numeric types have corner cases that do not behave like the integers and real numbers we're used to. As a result, some errors that really should be dynamically checked are not checked at all. Here are the traps:

- **Integer division** . `5/2` does not return a fraction, it returns a truncated integer. So this is an example of where what we might have hoped would be a dynamic error (because a fraction isn't representable as an integer) frequently produces the wrong answer instead.
- **Integer overflow** . The `int` and `long` types are actually finite sets of integers, with maximum and minimum values. What happens when you do a computation whose answer is too positive or too negative to fit in that finite range? The computation quietly *overflows* (wraps around), and returns an integer from somewhere in the legal range but not the right answer.
- **Special values in float and doubles** . The `float` and `double` types have several special values that aren't real numbers: `NaN` (which stands for "Not a Number"), `POSITIVE_INFINITY` , and `NEGATIVE_INFINITY` . So operations that you'd expect to produce dynamic errors, like dividing by zero or taking the square root of a negative number, produce one of these special values instead. If you keep computing with it, you'll end up with a bad final answer.

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`
- length: `a.length` (note that this is different syntax from `String.length()` – `a.length` is not a method call, so you don't put parentheses after it)

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]; // ===== DANGER WILL ROBINSON
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` that turned out to have a very long hailstone sequence? It wouldn't fit in a length-100 array. 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. Here's how we can declare a `List` variable and make a list value:

```
List<Integer> list = new ArrayList<Integer>();
```

And here are some of its operations:

- indexing: `list.get(2)`
- assignment: `list.set(2, 0)`
- length: `list.size()`

Note that `List` is an interface, a type that can't be constructed directly with `new`, but that instead specifies the operations that a `List` must provide. We'll talk about this notion in a future class 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, which you can find by searching the web for "Java 8 API". Get to know the Java API docs, they're your friend. ("API" means "application programmer interface," and is commonly used as a synonym for "library.")

Note also that we wrote `List<Integer>` instead of `List<int>`. Unfortunately we 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 (which are written in lowercase and often abbreviated, like `int`) has an equivalent object type (which is capitalized, and fully spelled out, like `Integer`). Java requires us to use these object type equivalents when we parameterize a type with angle brackets. But in other contexts, 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 Lists:

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

Not only simpler but safer too, because the `List` automatically enlarges itself to fit as many numbers as you add to it (until you run out of memory, of course).

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:

```
// find the maximum point of a hailstone sequence stored in list
int max = 0;
for (int x : list) {
    max = Math.max(x, max);
}
```

You can iterate through arrays as well as lists. The same code would work if the list were replaced by an array.

`Math.max()` is a handy function from the Java API. The `Math` class is full of useful functions like this – search for "java 8 Math" on the web to find its documentation.

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

```

Let's explain a few of the new things here.

`public` means that any code, anywhere in your program, can refer to the class or method. 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 class.

`static` means that the method doesn't take a self parameter – which in Java is implicit anyway, you won't ever see it as a method parameter. Static methods can't be called on an object. Contrast that with the `List add()` method or the `String length()` method, for example, which require an object to come first. Instead, the right way to call a static method uses the class name instead of an object reference:

```
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 positive, which is not captured by the type declaration but is very important for the caller to know.

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

Mutating Values vs. Reassigning Variables

The next reading will introduce *snapshot diagrams* to 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 in a future class 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> list = 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 (unfortunately) doesn't check automatically: that `n` must be positive.

Why do we need to write down our assumptions? Because 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. First persuading the compiler that your program is sensible – syntactically correct and type-correct. Then getting the logic right so that it gives the right results at runtime.
- communicating with other people. Making the program easy to understand, so that when somebody has to fix it, improve it, or adapt it in the future, they can do so.

Hacking vs. Engineering

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

- Bad: writing lots of code before testing any of it
- Bad: keeping all the details in your head, assuming you'll remember them forever, instead of writing them down in your code

- Bad: assuming that bugs will be nonexistent or else easy to find and fix

But software engineering is not hacking. Engineers are pessimists:

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

The Goal of 6.005

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 write 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.

There are other important properties of software (like performance, usability, security), and they may trade off against these three. But these are the Big Three that we care about in 6.005, and that software developers generally put foremost in the practice of building software. 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.

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, but other kinds of static checks too, like that your code returns values from methods declared to do so). 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 how in a safe, statically-checked language.

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 certainly 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, test frameworks, profilers, code coverage, style checkers). 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, having accumulated many features over the years. It's internally inconsistent (e.g. the `final` keyword means different things in different contexts, and the `static` keyword in Java has nothing to do with static checking). It's weighted with the baggage of older languages like C/C++ (the primitive types and the `switch` statement are good examples). It has no interpreter like Python's, where you can learn by playing with small bits of code.

But on the whole, Java is a reasonable choice of language right now to learn how to write code that is safe from bugs, easy to understand, and ready for change. And that's our goal.

Summary

The main idea we introduced today is **static checking**. Here's how this idea relates to the goals of the course:

- **Safe from bugs.** Static checking helps with safety by catching type errors and other bugs before runtime.
- **Easy to understand.** It helps with understanding, because types are explicitly stated in the code.
- **Ready for change.** Static checking makes it easier to change your code by identifying other places that need to change in tandem. For example, when you change the name or type of a variable, the compiler immediately displays errors at all the places where that variable is used, reminding you to update them as well.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

Reading 2: Basic Java

Objectives

- Learn basic Java syntax and semantics
- Transition from writing Python to writing Java

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Getting started with the Java Tutorials

The next few sections link to the [Java Tutorials](https://docs.oracle.com/javase/tutorial/index.html) ([//docs.oracle.com/javase/tutorial/index.html](https://docs.oracle.com/javase/tutorial/index.html)) to get you up to speed with the basics.

You can also look at Getting Started: Learning Java ([..../getting-started/java.html](https://docs.oracle.com/javase/tutorial/getStarted/intro/index.html)) as an alternative resource.

This reading and other resources will frequently refer you to the Java API documentation ([//docs.oracle.com/javase/8/docs/api/](https://docs.oracle.com/javase/8/docs/api/)) which describes all the classes built in to Java.

Language basics

Read [Language Basics](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html) ([//docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html)) .

You should be able to answer the questions on the *Questions and Exercises* pages for all four of the language basics topics.

- Questions: Variables
([//docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_variables.html](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_variables.html))
- Questions: Operators
([//docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_operators.html](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_operators.html))
- Questions: Expressions, Statements, Blocks
([//docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_expressions.html](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_expressions.html))

- Questions: Control Flow
https://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_flow.html

Note that each *Questions and Exercises* page has a link at the bottom to solutions.

Also check your understanding by answering some questions about how the basics of Java compare to the basics of Python:

Numbers and strings

Read **Numbers and Strings** (<https://docs.oracle.com/javase/tutorial/java/data/index.html>) .

Don't worry if you find the `Number` wrapper classes confusing. They are.

You should be able to answer the questions on both *Questions and Exercises* pages.

- Questions: Numbers (<https://docs.oracle.com/javase/tutorial/java/data/QandE/numbers-questions.html>)
- Questions: Characters, Strings (<https://docs.oracle.com/javase/tutorial/java/data/QandE/characters-questions.html>)

Classes and objects

Read **Classes and Objects** (<https://docs.oracle.com/javase/tutorial/java/javaOO/index.html>) .

You should be able to answer the questions on the first two *Questions and Exercises* pages.

- Questions: Classes (<https://docs.oracle.com/javase/tutorial/java/javaOO/QandE/creating-questions.html>)
- Questions: Objects (<https://docs.oracle.com/javase/tutorial/java/javaOO/QandE/objects-questions.html>)

Don't worry if you don't understand everything in *Nested Classes* and *Enum Types* right now. You can go back to those constructs later in the semester when we see them in class.

Hello, world!

Read **Hello World!** (<https://docs.oracle.com/javase/tutorial/getStarted/application/index.html>)

You should be able to create a new `HelloWorldApp.java` file, enter the code from that tutorial page, and compile and run the program to see `Hello World!` on the console.

Snapshot diagrams

It will be useful for us to draw pictures of what's happening at runtime, in order to understand subtle questions. **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 class and in 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 (with each other and with your TA).

- To pave the way for richer design notations in subsequent courses. For example, snapshot diagrams generalize into object models in 6.170.

Although the diagrams in this course use examples from Java, the notation can be applied to any modern programming language, e.g., Python, Javascript, C++, Ruby.

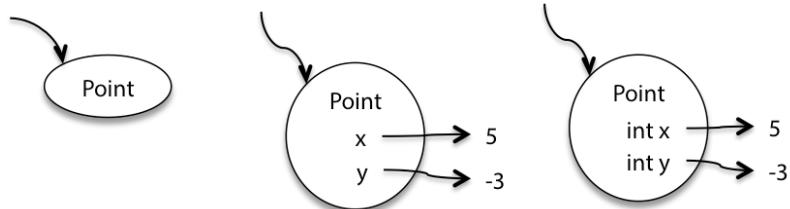
Primitive values

Primitive values are represented by bare constants. The incoming arrow is a reference to the value 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`, but both are fine.

Mutating values vs. reassigning variables

Snapshot diagrams give us a way to visualize the distinction between changing a variable and changing a value:

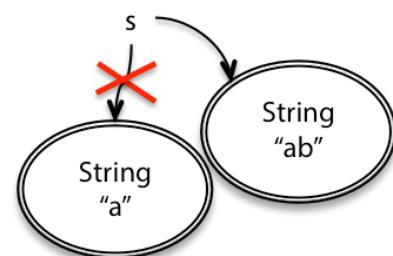
- When you assign to a variable or a field, you're changing where the variable's arrow points. You can point it to a different value.
- When you assign to the contents of a mutable value – such as an array or list – you're changing references inside that value.

Reassignment and immutable values

For example, if we have a `String`

(//docs.oracle.com/javase/8/docs/api/?java/lang/String.html) variable `s`, we can reassign it from a value of "a" to "ab".

```
String s = "a";
s = s + "b";
```



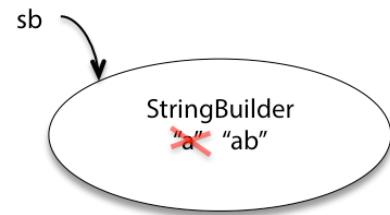
`String` is an example of an *immutable* type, a type whose values can never change once they have been created. Immutability (immunity from change) is a major design principle in this course, and we'll talk much more about it in future readings.

Immutable objects (intended by their designer to always represent the same value) are denoted in a snapshot diagram by a double border, like the `String` objects in our diagram.

Mutable values

By contrast, `StringBuilder` (//docs.oracle.com/javase/8/docs/api/?java/lang/StringBuilder.html) (another built-in Java class) is a *mutable* object that represents a string of characters, and it has methods that change the value of the object:

```
StringBuilder sb = new StringBuilder("a");
sb.append("b");
```



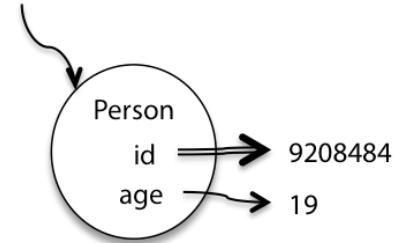
These two snapshot diagrams look very different, which is good: the difference between mutability and immutability will play an important role in making our code safe from bugs .

Immutable references

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.



In a snapshot diagram, an immutable reference (`final`) is denoted by a double arrow. Here's an object whose `id` never changes (it can't be reassigned to a different number), but whose `age` can change.

Notice that we can have an *immutable reference* to a *mutable value* (for example: `final StringBuilder sb`) whose value can change even though we're pointing to the same object.

We can also have a *mutable reference* to an *immutable value* (like `String s`), where the value of the variable can change because it can be re-pointed to a different object.

Java Collections

The very first Language Basics tutorial discussed **arrays**

([//docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html)), which are *fixed-length* containers for a sequence of objects or primitive values. Java provides a number of more powerful and flexible tools for managing *collections* of objects: the **Java Collections Framework** .

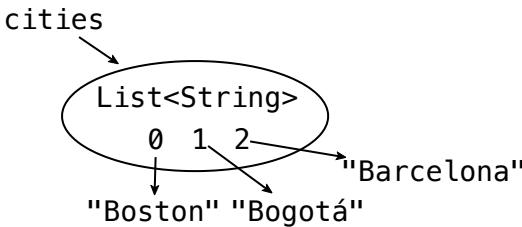
Lists, Sets, and Maps

A Java `List` ([//docs.oracle.com/javase/8/docs/api/?java/util/List.html](https://docs.oracle.com/javase/8/docs/api/?java/util/List.html)) is similar to a Python `list` (<https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range>) . A `List` contains an ordered collection of zero or more objects, where the same object might appear multiple times. We can add and remove items to and from the `List` , which will grow and shrink to accomodate its contents.

Example `List` operations:

Java	description	Python
<code>int count = lst.size();</code>	count the number of elements	<code>count = len(lst)</code>
<code>lst.add(e);</code>	append an element to the end	<code>lst.append(e)</code>
<code>if (lst.isEmpty()) ...</code>	test if the list is empty	<code>if not lst: ...</code>

In a snapshot diagram, we represent a `List` as an object with indices drawn as fields:



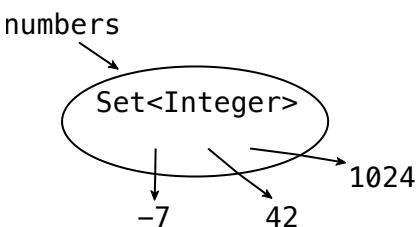
This list of `cities` might represent a trip from Boston to Bogotá to Barcelona.

A Set (<https://docs.oracle.com/javase/8/docs/api/?java/util/Set.html>) is an unordered collection of zero or more unique objects. Like a mathematical set or a Python set (<https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset>) – and unlike a `List` – an object cannot appear in a set multiple times. Either it's in or it's out.

Example Set operations:

Java	description	Python
<code>s1.contains(e)</code>	test if the set contains an element	<code>e in s1</code>
<code>s1.containsAll(s2)</code>	test whether $s1 \supseteq s2$	<code>s1.issuperset(s2)</code> <code>s1 >= s2</code>
<code>s1.removeAll(s2)</code>	remove $s2$ from $s1$	<code>s1.difference_update(s2)</code> <code>s1 -= s2</code>

In a snapshot diagram, we represent a `Set` as an object with no-name fields:



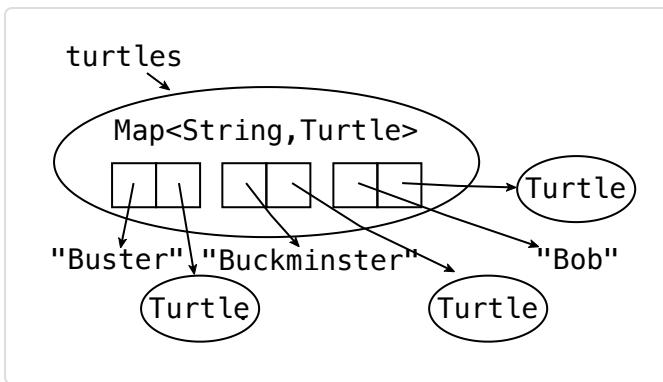
Here we have a set of integers, in no particular order: 42, 1024, and -7.

A Map (<https://docs.oracle.com/javase/8/docs/api/?java/util/Map.html>) is similar to a Python dictionary (<https://docs.python.org/3/library/stdtypes.html#mapping-types-dict>). In Python, the **keys** of a map must be hashable (<https://docs.python.org/3/glossary.html#term-hashable>). Java has a similar requirement that we'll discuss when we confront how equality works between Java objects.

Example Map operations:

Java	description	Python
<code>map.put(key, val)</code>	add the mapping $key \rightarrow val$	<code>map[key] = val</code>
<code>map.get(key)</code>	get the value for a key	<code>map[key]</code>
<code>map.containsKey(key)</code>	test whether the map has a key	<code>key in map</code>
<code>map.remove(key)</code>	delete a mapping	<code>del map[key]</code>

In a snapshot diagram, we represent a `Map` as an object that contains key/value pairs:



This `turtles` map contains `Turtle` objects assigned to `String` keys: `Bob`, `Buckminster`, and `Buster`.

Literals

Python provides convenient syntax for creating lists:

```
lst = [ "a", "b", "c" ]
```

And maps:

```
map = { "apple": 5, "banana": 7 }
```

Java does not. It does provide a literal syntax for arrays:

```
String[] arr = { "a", "b", "c" };
```

But this creates an `array`, not a `List`. We can use a provided utility function ([//docs.oracle.com/javase/8/docs/api/?java/util/Arrays.html](https://docs.oracle.com/javase/8/docs/api/?java/util/Arrays.html)) to create a `List` from the array:

```
Arrays.asList(new String[] { "a", "b", "c" })
```

A `List` created with `Arrays.asList` does come with a restriction: its length is fixed.

Generics: declaring List, Set, and Map variables

Unlike Python collection types, with Java collections we can restrict the type of objects contained in the collection. When we add an item, the compiler can perform *static checking* to ensure we only add items of the appropriate type. Then, when we pull out an item, we are guaranteed that its type will be what we expect.

Here's the syntax for declaring some variables to hold collections:

```
List<String> cities;           // a List of Strings
Set<Integer> numbers;         // a Set of Integers
Map<String,Turtle> turtles;   // a Map with String keys and Turtle values
```

Because of the way generics work, we cannot create a collection of primitive types. For example, `Set<int>` does *not* work. However, as we saw earlier, `int`'s have an `Integer` wrapper we can use (e.g. `Set<Integer> numbers`).

In order to make it easier to use collections of these wrapper types, Java does some automatic conversion. If we have declared `List<Integer> sequence`, this code works:

```
sequence.add(5);           // add 5 to the sequence
int second = sequence.get(1); // get the second element
```

ArrayLists and LinkedLists: creating Lists

As we'll see soon enough, Java helps us distinguish between the *specification* of a type – what does it do? – and the *implementation* – what is the code?

`List`, `Set`, and `Map` are all *interfaces*: they define how these respective types work, but they don't provide implementation code. There are several advantages, but one potential advantage is that we, the users of these types, get to choose different implementations in different situations.

Here's how to create some actual `List`s:

```
List<String> firstNames = new ArrayList<String>();
List<String> lastNames = new LinkedList<String>();
```

If the generic type parameters are the same on the left and right, Java can infer what's going on and save us some typing:

```
List<String> firstNames = new ArrayList<>();
List<String> lastNames = new LinkedList<>();
```

`ArrayList` ([//docs.oracle.com/javase/8/docs/api/?java/util/ArrayList.html](https://docs.oracle.com/javase/8/docs/api/?java/util/ArrayList.html)) and `LinkedList` ([//docs.oracle.com/javase/8/docs/api/?java/util/LinkedList.html](https://docs.oracle.com/javase/8/docs/api/?java/util/LinkedList.html)) are two implementations of `List`. Both provide all the operations of `List`, and those operations must work as described in the documentation for `List`. In this example, `firstNames` and `lastNames` will behave the same way; if we swapped which one used `ArrayList` vs. `LinkedList`, our code will not break.

Unfortunately, this ability to choose is also a burden: we didn't care how Python lists worked, why should we care whether our Java lists are `ArrayLists` or `LinkedLists`? Since the only difference is performance, for 6.005 we *don't*.

When in doubt, use `ArrayList`.

HashSets and HashMaps: creating Sets and Maps

`HashSet` ([//docs.oracle.com/javase/8/docs/api/?java/util/HashSet.html](https://docs.oracle.com/javase/8/docs/api/?java/util/HashSet.html)) is our default choice for `Set`s:

```
Set<Integer> numbers = new HashSet<>();
```

Java also provides sorted sets ([//docs.oracle.com/javase/8/docs/api/?java/util/SortedSet.html](https://docs.oracle.com/javase/8/docs/api/?java/util/SortedSet.html)) with the `TreeSet` ([//docs.oracle.com/javase/8/docs/api/?java/util/TreeSet.html](https://docs.oracle.com/javase/8/docs/api/?java/util/TreeSet.html)) implementation.

And for a `Map` the default choice is `HashMap` ([//docs.oracle.com/javase/8/docs/api/?java/util/HashMap.html](https://docs.oracle.com/javase/8/docs/api/?java/util/HashMap.html)):

```
Map<String, Turtle> turtles = new HashMap<>();
```

Iteration

So maybe we have:

```
List<String> cities      = new ArrayList<>();
Set<Integer> numbers    = new HashSet<>();
Map<String,Turtle> turtles = new HashMap<>();
```

A very common task is iterating through our cities/numbers/turtles/etc.

In Python:

```
for city in cities:
    print city

for num in numbers:
    print num

for key in turtles:
    print "%s: %s" % (key, turtles[key])
```

Java provides a similar syntax for iterating over the items in `List`'s and `Set`'s.

Here's the Java:

```
for (String city : cities) {
    System.out.println(city);
}

for (int num : numbers) {
    System.out.println(num);
}
```

We can't iterate over `Map`'s themselves this way, but we can iterate over the keys as we did in Python:

```
for (String key : turtles.keySet()) {
    System.out.println(key + ": " + turtles.get(key));
}
```

Under the hood this kind of `for` loop uses an `Iterator` ([//docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html](https://docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html)) , a design pattern we'll see later in the class.

Iterating with indices

If you want to, Java provides different `for` loops that we can use to iterate through a list using its indices:

```
for (int ii = 0; ii < cities.size(); ii++) {
    System.out.println(cities.get(ii));
}
```

Unless we actually need the index value `ii` , this code is verbose and has more places for bugs to hide. Avoid.

Java API documentation

The previous section has a number of links to documentation for classes that are part of the Java platform API ([//docs.oracle.com/javase/8/docs/api/](https://docs.oracle.com/javase/8/docs/api/)).

API stands for *application programming interface*. If you want to program an app that talks to Facebook, Facebook publishes an API (more than one, in fact, for different languages and frameworks) you can program against. The Java API is a large set of generally useful tools for programming pretty much anything.

- **java.lang.String** ([//docs.oracle.com/javase/8/docs/api/?java/lang/String.html](https://docs.oracle.com/javase/8/docs/api/?java/lang/String.html)) is the full name for `String`. We can create objects of type `String` just by using "double quotes".
- **java.lang.Integer** ([//docs.oracle.com/javase/8/docs/api/?java/lang/Integer.html](https://docs.oracle.com/javase/8/docs/api/?java/lang/Integer.html)) and the other primitive wrapper classes. Java automatically converts between primitive and wrapped (or "boxed") types in most situations.
- **java.util.List** ([//docs.oracle.com/javase/8/docs/api/?java/util/List.html](https://docs.oracle.com/javase/8/docs/api/?java/util/List.html)) is like a Python list, but in Python, lists are part of the language. In Java, `List`s are implemented in... Java!
- **java.util.Map** ([//docs.oracle.com/javase/8/docs/api/?java/util/Map.html](https://docs.oracle.com/javase/8/docs/api/?java/util/Map.html)) is like a Python dictionary.
- **java.io.File** ([//docs.oracle.com/javase/8/docs/api/?java/io/File.html](https://docs.oracle.com/javase/8/docs/api/?java/io/File.html)) represents a file on disk. Take a look at the methods provided by `File`: we can test whether the file is readable, delete the file, see when it was last modified...
- **java.io.FileReader** ([//docs.oracle.com/javase/8/docs/api/?java/io/FileReader.html](https://docs.oracle.com/javase/8/docs/api/?java/io/FileReader.html)) lets us read text files.
- **java.io.BufferedReader** ([//docs.oracle.com/javase/8/docs/api/?java/io/BufferedReader.html](https://docs.oracle.com/javase/8/docs/api/?java/io/BufferedReader.html)) lets us read in text efficiently, and it also provides a very useful feature: reading an entire line at a time.

Let's take a closer look at the documentation for `BufferedReader`

([//docs.oracle.com/javase/8/docs/api/?java/io/BufferedReader.html](https://docs.oracle.com/javase/8/docs/api/?java/io/BufferedReader.html)). There are many things here that relate to features of Java we haven't discussed! Keep your head and focus on the **things in bold** below.

At the top of the page is the *class hierarchy* for `BufferedReader` and a list of *implemented interfaces*. A `BufferedReader` object has all of the methods of all those types (plus its own methods) available to use.

Next we see *direct subclasses*, and for an interface, *implementing classes*. This can help us find, for example, that `HashMap` ([//docs.oracle.com/javase/8/docs/api/?java/util/HashMap.html](https://docs.oracle.com/javase/8/docs/api/?java/util/HashMap.html)) is an implementation of `Map` ([//docs.oracle.com/javase/8/docs/api/?java/util/Map.html](https://docs.oracle.com/javase/8/docs/api/?java/util/Map.html)).

Next up: a **description of the class**. Sometimes these descriptions are a little obtuse, but **this is the first place you should go** to understand a class.

If you want to make a new `BufferedReader` the **constructor summary** is the first place to look.

The screenshot shows the Java API documentation for the `BufferedReader` class. At the top, there's a navigation bar with links for Overview, Package, Class (which is highlighted), Use, Tree, Deprecated, Index, and Help. Below the navigation bar, there are links for PREV CLASS, NEXT CLASS, FRAMES, NO FRAMES, and ALL CLASSES. The summary section includes links for SUMMARY: NESTED | FIELD | CONSTR | METHOD and DETAIL: FIELD | CONSTR | METHOD. The main content area starts with the `compact1`, `compact2`, and `compact3` methods from the `java.io` package. Then it defines the `Class BufferedReader`, which extends `java.lang.Object` and implements `java.io.Reader` and `java.io.BufferedReader`. It lists `Closeable`, `AutoCloseable`, and `Readable` as implemented interfaces, and `LineNumberReader` as a direct known subclass. The detailed description explains that `BufferedReader` reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. It notes that the buffer size may be specified, or the default size may be used. The buffer size is large enough for most purposes. It also states that in general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a `BufferedReader` around any Reader whose `read()` operations may be costly, such as `FileReaders` and `InputStreamReaders`. For example:

```
BufferedReader in
= new BufferedReader(new FileReader("foo.in"));
```

This will buffer the input from the specified file. Without buffering, each invocation of `read()` or `readLine()` could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient.

Programs that use `DataInputStreams` for textual input can be localized by replacing each `DataInputStream` with an appropriate `BufferedReader`.

Constructor Summary

Constructors

Constructor and Description

`BufferedReader(Reader in)`
Creates a buffering character-input stream that uses a default-sized input buffer.

`BufferedReader(Reader in, int sz)`
Creates a buffering character-input stream that uses an input buffer of the specified size.

Constructors aren't the only way to get a new object in Java, but they are the most common.

Next: the method summary lists all the methods we can call on a `BufferedReader` object.

Below the summary are detailed descriptions of each method and constructor. **Click a constructor or method to see the detailed description.** This is the first place you should go to understand what a method does.

Method Summary		
All Methods	Instance Methods	Concrete Methods
Modifier and Type	Method and Description	
void	<code>close()</code> Closes the stream and releases any system resources associated with it.	
<code>Stream<String></code>	<code>lines()</code> Returns a Stream, the elements of which are lines read from this BufferedReader.	
void	<code>mark(int readAheadLimit)</code> Marks the present position in the stream.	
boolean	<code>markSupported()</code> Tells whether this stream supports the mark() operation, which it does.	
int	<code>read()</code> Reads a single character.	
int	<code>read(char[] cbuf, int off, int len)</code> Reads characters into a portion of an array.	
<code>String</code>	<code>readLine()</code> Reads a line of text.	
boolean	<code>ready()</code> Tells whether this stream is ready to be read.	
void	<code>reset()</code> Resets the stream to the most recent mark.	
long	<code>skip(long n)</code> Skips characters.	
Methods inherited from class <code>java.io.Reader</code>		
<code>read(), read()</code>		
Methods inherited from class <code>java.lang.Object</code>		
<code>clone(), equals(), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(), wait()</code>		

Each detailed description includes:

- The **method signature** : we see the return type, the method name, and the parameters. We also see *exceptions* . For now, those usually mean errors the method can run into.
- The full **description** .
- **Parameters** : descriptions of the method arguments.
- And a description of what the method **returns** .

Specifications

These detailed descriptions are **specifications** . They allow us to use tools like `String` , `Map` , or `BufferedReader` *without* having to read or understand the code that implements them.

Reading, writing, understanding, and analyzing specifications will be one of our first major undertakings in 6.005, starting in a few classes.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 3: Testing

Validation

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Documenting Your Testing Strategy

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

Reading 3: Testing

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives for Today's Class

After today's class, you should:

- understand the value of testing, and know the process of test-first programming;
- be able to design a test suite for a method by partitioning its input and output space and choosing good test cases;
- be able to judge a test suite by measuring its code coverage; and
- understand and know when to use blackbox vs. whitebox testing, unit tests vs. integration tests, and automated regression testing.

Validation

Testing is an example of a more general process called *validation*. The purpose of validation is to uncover problems in a program and thereby increase your confidence in the program's correctness. Validation includes:

- **Formal reasoning** about a program, usually called *verification*. Verification constructs a formal proof that a program is correct. Verification is tedious to do by hand, and automated tool support for verification is still an active area of research. Nevertheless, small, crucial pieces of a program may be formally verified, such as the scheduler in an operating system, or the bytecode interpreter

- in a virtual machine, or the filesystem in an operating system ([//www.csail.mit.edu/crash_tolerant_data_storage](http://www.csail.mit.edu/crash_tolerant_data_storage)) .
- **Code review.** Having somebody else carefully read your code, and reason informally about it, can be a good way to uncover bugs. It's much like having somebody else proofread an essay you have written. We'll talk more about code review in the next reading.
 - **Testing** . Running the program on carefully selected inputs and checking the results.

Even with the best validation, it's very hard to achieve perfect quality in software. Here are some typical *residual defect rates* (bugs left over after the software has shipped) per kloc (one thousand lines of source code):

- 1 - 10 defects/kloc: Typical industry software.
- 0.1 - 1 defects/kloc: High-quality validation. The Java libraries might achieve this level of correctness.
- 0.01 - 0.1 defects/kloc: The very best, safety-critical validation. NASA and companies like Praxis can achieve this level.

This can be discouraging for large systems. For example, if you have shipped a million lines of typical industry source code (1 defect/kloc), it means you missed 1000 bugs!

Why Software Testing is Hard

Here are some approaches that unfortunately don't work well in the world of software.

Exhaustive testing is infeasible. The space of possible test cases 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!

Haphazard testing (“just try it and see if it works”) is less likely to find bugs, unless the program is so buggy that an arbitrarily-chosen input is more likely to fail than to succeed. It also doesn’t increase our confidence in program correctness.

Random or statistical testing doesn’t work well for software. Other engineering disciplines can test small random samples (e.g. 1% of hard drives manufactured) and infer the defect rate for the whole production lot. Physical systems can use many tricks to speed up time, like opening a refrigerator 1000 times in 24 hours instead of 10 years. These tricks give known failure rates (e.g. mean lifetime of a hard drive), but they assume continuity or uniformity across the space of defects. This is true for physical artifacts.

But it’s not true for software. Software behavior varies discontinuously and discretely across the space of possible inputs. The system may seem to work fine across a broad range of inputs, and then abruptly fail at a single boundary point. The famous Pentium division bug ([//www.willamette.edu/~mjaneba/pentprob.html](http://www.willamette.edu/~mjaneba/pentprob.html)) affected approximately 1 in 9 billion divisions. Stack overflows, out of memory errors, and numeric overflow bugs tend to happen abruptly, and always in the same way, not with probabilistic variation. That’s different from physical systems, where there is often visible evidence that the system is approaching a failure point (cracks in a bridge) or failures are distributed probabilistically near the failure point (so that statistical testing will observe some failures even before the point is reached).

Instead, test cases must be chosen carefully and systematically, and that’s what we’ll look at next.

Putting on Your Testing Hat

Testing requires having the right attitude. When you’re coding, your goal is to make the program work, but as a tester, you want to **make it fail** .

That's a subtle but important difference. It is all too tempting to treat code you've just written as a precious thing, a fragile eggshell, and test it very lightly just to see it work.

Instead, you have to be brutal. A good tester wields a sledgehammer and beats the program everywhere it might be vulnerable, so that those vulnerabilities can be eliminated.

Test-first Programming

Test early and often. Don't leave testing until the end, when you have a big pile of unvalidated code. Leaving testing until the end only makes debugging longer and more painful, because bugs may be anywhere in your code. It's far more pleasant to test your code as you develop it.

In test-first-programming, you write tests before you even write any code. The development of a single function proceeds in this order:

1. Write a specification for the function.
2. Write tests that exercise the specification.
3. Write the actual code. Once your code passes the tests you wrote, you're done.

The **specification** describes the input and output behavior of the function. It gives the types of the parameters and any additional constraints on them (e.g. `sqrt`'s parameter must be nonnegative). It also gives the type of the return value and how the return value relates to the inputs. You've already seen and used specifications on your problem sets in this class. In code, the specification consists of the method signature and the comment above it that describes what it does. We'll have much more to say about specifications a few classes from now.

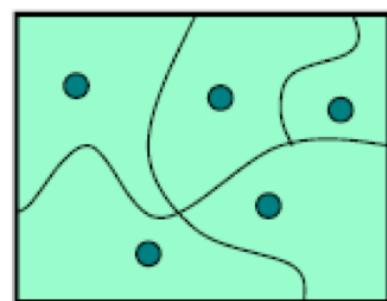
Writing tests first is a good way to understand the specification. The specification can be buggy, too — incorrect, incomplete, ambiguous, missing corner cases. Trying to write tests can uncover these problems early, before you've wasted time writing an implementation of a buggy spec.

Choosing Test Cases by Partitioning

Creating a good test suite is a challenging and interesting design problem. We want to pick a set of test cases that is small enough to run quickly, yet large enough to validate the program.

To do this, we divide the input space into **subdomains**, each consisting of a set of inputs. Taken together the subdomains completely cover the input space, so that every input lies in at least one subdomain. Then we choose one test case from each subdomain, and that's our test suite.

The idea behind subdomains is to partition the input space into sets of similar inputs on which the program has similar behavior. Then we use one representative of each set. This approach makes the best use of limited testing resources by choosing dissimilar test cases, and forcing the testing to explore parts of the input space that random testing might not reach.



We can also partition the output space into subdomains (similar outputs on which the program has similar behavior) if we need to ensure our tests will explore different parts of the output space. Most of the time, partitioning the input space is sufficient.

Example: `BigInteger.multiply()`

Let's look at an example. `BigInteger` ([//docs.oracle.com/javase/8/docs/api/?java/math/BigInteger.html](https://docs.oracle.com/javase/8/docs/api/?java/math/BigInteger.html)) is a class built into the Java library that can represent integers of any size, unlike the primitive types `int` and `long` that have only limited ranges. `Biglnteger` has a method

`multiply` that multiplies two BigInteger values together:

```
/**  
 * @param val another BigInteger  
 * @return a BigInteger whose value is (this * val).  
 */  
public BigInteger multiply(BigInteger val)
```

For example, here's how it might be used:

```
BigInteger a = ...;  
BigInteger b = ...;  
BigInteger ab = a.multiply(b);
```

This example shows that even though only one parameter is explicitly shown in the method's declaration, `multiply` is actually a function of *two* arguments: the object you're calling the method on (`a` in the example above), and the parameter that you're passing in the parentheses (`b` in this example). In Python, the object receiving the method call would be explicitly named as a parameter called `self` in the method declaration. In Java, you don't mention the receiving object in the parameters, and it's called `this` instead of `self`.

So we should think of `multiply` as a function taking two inputs, each of type `BigInteger`, and producing one output of type `BigInteger`:

`multiply : BigInteger × BigInteger → BigInteger`

So we have a two-dimensional input space, consisting of all the pairs of integers (a,b). Now let's partition it. Thinking about how multiplication works, we might start with these partitions:

- a and b are both positive
- a and b are both negative
- a is positive, b is negative
- a is negative, b is positive

There are also some special cases for multiplication that we should check: 0, 1, and -1.

- a or b is 0, 1, or -1

Finally, as a suspicious tester trying to find bugs, we might suspect that the implementor of `BigInteger` might try to make it faster by using `int` or `long` internally when possible, and only fall back to an expensive general representation (like a list of digits) when the value is too big. So we should definitely also try integers that are very big, bigger than the biggest `long`.

- a or b is small
- the absolute value of a or b is bigger than `Long.MAX_VALUE`, the biggest possible primitive integer in Java, which is roughly 2^{63} .

Let's bring all these observations together into a straightforward partition of the whole (a,b) space. We'll choose a and b independently from:

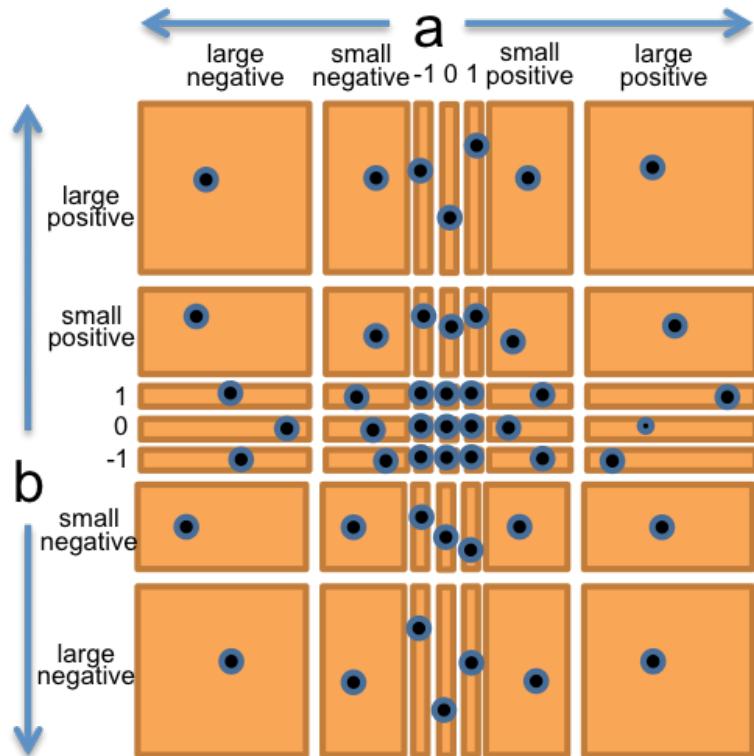
- 0
- 1
- -1
- small positive integer
- small negative integer
- huge positive integer
- huge negative integer

So this will produce $7 \times 7 = 49$ partitions that completely cover the space of pairs of integers.

To produce the test suite, we would pick an arbitrary pair (a,b) from each square of the grid, for example:

- $(a,b) = (-3, 25)$ to cover (small negative, small positive)
- $(a,b) = (0, 30)$ to cover (0, small positive)
- $(a,b) = (2^{100}, 1)$ to cover (large positive, 1)
- etc.

The figure at the right shows how the two-dimensional (a,b) space is divided by this partition, and the points are test cases that we might choose to completely cover the partition.



Example: `max()`

Let's look at another example from the Java library: the integer `max()` function, found in the `Math` class ([//docs.oracle.com/javase/8/docs/api/java/lang/Math.html](https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html)).

```
/**
 * @param a  an argument
 * @param b  another argument
 * @return the larger of a and b.
 */
public static int max(int a, int b)
```

Mathematically, this method is a function of the following type:

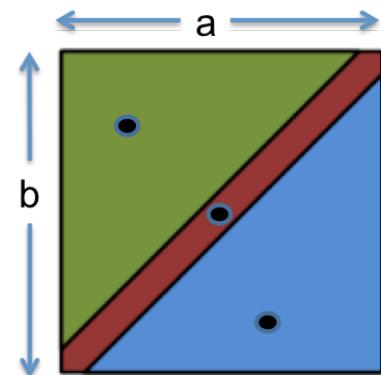
`max : int × int → int`

From the specification, it makes sense to partition this function as:

- $a < b$
- $a = b$
- $a > b$

Our test suite might then be:

- $(a, b) = (1, 2)$ to cover $a < b$
- $(a, b) = (9, 9)$ to cover $a = b$
- $(a, b) = (-5, -6)$ to cover $a > b$



Include Boundaries in the Partition

Bugs often occur at *boundaries* between subdomains. Some examples:

- 0 is a boundary between positive numbers and negative numbers

- the maximum and minimum values of numeric types, like `int` and `double`
- emptiness (the empty string, empty list, empty array) for collection types
- the first and last element of a collection

Why do bugs often happen at boundaries? One reason is that programmers often make **off-by-one mistakes** (like writing `<=` instead of `<`, or initializing a counter to 0 instead of 1). Another is that some boundaries may need to be handled as special cases in the code. Another is that boundaries may be places of discontinuity in the code's behavior. When an `int` variable grows beyond its maximum positive value, for example, it abruptly becomes a negative number.

It's important to include boundaries as subdomains in your partition, so that you're choosing an input from the boundary.

Let's redo `max : int × int → int`.

Partition into:

- *relationship between a and b*
 - $a < b$
 - $a = b$
 - $a > b$
- *value of a*
 - $a = 0$
 - $a < 0$
 - $a > 0$
 - $a = \text{minimum integer}$
 - $a = \text{maximum integer}$
- *value of b*
 - $b = 0$
 - $b < 0$
 - $b > 0$
 - $b = \text{minimum integer}$
 - $b = \text{maximum integer}$

Now let's pick test values that cover all these classes:

- (1, 2) covers $a < b$, $a > 0$, $b > 0$
- (-1, -3) covers $a > b$, $a < 0$, $b < 0$
- (0, 0) covers $a = b$, $a = 0$, $b = 0$
- (`Integer.MIN_VALUE`, `Integer.MAX_VALUE`) covers $a < b$, $a = \text{minint}$, $b = \text{maxint}$
- (`Integer.MAX_VALUE`, `Integer.MIN_VALUE`) covers $a > b$, $a = \text{maxint}$, $b = \text{minint}$

Two Extremes for Covering the Partition

After partitioning the input space, we can choose how exhaustive we want the test suite to be:

- **Full Cartesian product.**

Every legal combination of the partition dimensions is covered by one test case. This is what we did for the `multiply` example, and it gave us $7 \times 7 = 49$ test cases. For the `max` example that included boundaries, which has three dimensions with 3 parts, 5 parts, and 5 parts respectively, it would mean up to $3 \times 5 \times 5 = 75$ test cases. In practice not all of these combinations are possible, however. For example, there's no way to cover the combination $a < b$, $a=0$, $b=0$, because `a` can't be simultaneously less than zero and equal to zero.

- **Cover each part.**

Every part of each dimension is covered by at least one test case, but not necessarily every combination. With this approach, the test suite for `max` might be as small as 5 test cases if

carefully chosen. That's the approach we took above, which allowed us to choose 5 test cases.

Often we strike some compromise between these two extremes, based on human judgement and caution, and influenced by whitebox testing and code coverage tools, which we look at next.

Blackbox and Whitebox Testing

Recall from above that the *specification* is the description of the function's behavior — the types of parameters, type of return value, and constraints and relationships between them.

Blackbox testing means choosing test cases only from the specification, not the implementation of the function. That's what we've been doing in our examples so far. We partitioned and looked for boundaries in `multiply` and `max` without looking at the actual code for these functions.

Whitebox testing (also called glass box testing) means choosing test cases with knowledge of how the function is actually implemented. For example, if the implementation selects different algorithms depending on the input, then you should partition according to those domains. If the implementation keeps an internal cache that remembers the answers to previous inputs, then you should test repeated inputs.

When doing whitebox testing, you must take care that your test cases don't *require* specific implementation behavior that isn't specifically called for by the spec. For example, if the spec says "throws an exception if the input is poorly formatted," then your test shouldn't check *specifically* for a `NullPointerException` just because that's what the current implementation does. The specification in this case allows *any* exception to be thrown, so your test case should likewise be general to preserve the implementor's freedom. We'll have much more to say about this in the class on specs.

Documenting Your Testing Strategy

For the example function on the left, on the right is how we can document the testing strategy we worked on in the partitioning exercises above. The strategy also addresses some boundary values we didn't consider before.

```
/**  
 * Reverses the end of a string.  
 *  
 * For example:  
 *   reverseEnd("Hello, world", 5)  
 *   returns "Hellodlrow ,"  
 *  
 * With start == 0, reverses the entire text.  
 * With start == text.length(), reverses nothing.  
 *  
 * @param text  non-null String that will have  
 *             its end reversed  
 * @param start the index at which the  
 *             remainder of the input is  
 *             reversed, requires  
 *             0 <= start <= text.length()  
 * @return input text with the substring from  
 *         start to the end of the string  
 *         reversed  
 */  
static String reverseEnd(String text, int start)
```

Document the strategy at the top of the test class:

```
/*  
 * Testing strategy  
 *  
 * Partition the inputs as follows:  
 * text.length(): 0, 1, > 1  
 * start:          0, 1, 1 < start < text.length(),  
 *                 text.length() - 1, text.length()  
 * text.length()-start: 0, 1, even > 1, odd > 1  
 *  
 * Include even- and odd-length reversals because  
 * only odd has a middle element that doesn't move.  
 *  
 * Exhaustive Cartesian coverage of partitions.  
 */
```

Document how each test case was chosen, including white box tests:

```
// covers text.length() = 0,  
//           start = 0 = text.length(),  
//           text.length()-start = 0  
@Test public void testEmpty() {  
    assertEquals("", reverseEnd("", 0));  
}  
  
// ... other test cases ...
```

Coverage

One way to judge a test suite is to ask how thoroughly it exercises the program. This notion is called **coverage**. Here are three common kinds of coverage:

- **Statement coverage**: is every statement run by some test case?
- **Branch coverage**: for every `if` or `while` statement in the program, are both the true and the false direction taken by some test case?
- **Path coverage**: is every possible combination of branches — every path through the program — taken by some test case?

Branch coverage is stronger (requires more tests to achieve) than statement coverage, and path coverage is stronger than branch coverage. In industry, 100% statement coverage is a common goal, but even that is rarely achieved due to unreachable defensive code (like “should never get here” assertions). 100% branch coverage is highly desirable, and safety critical industry code has even more arduous criteria (e.g., “MCDC,” modified decision/condition coverage). Unfortunately 100% path coverage is infeasible, requiring exponential-size test suites to achieve.

A standard approach to testing is to add tests until the test suite achieves adequate statement coverage: i.e., so that every reachable statement in the program is executed by at least one test case. In practice, statement coverage is usually measured by a code coverage tool, which counts the number of times each statement is run by your test suite. With such a tool, white box testing is easy; you just measure the coverage of your black box tests, and add more test cases until all important statements are logged as executed.

A good code

Element	Coverage	Covered Instructions	Total Instructions
multipart	6.7 %	48	717
src	6.7 %	48	717
multipart	0.0 %	0	75
sequence	57.8 %	48	83
FileSequenceReader.java	37.5 %	21	56
FileSequenceReaderTest.java	100.0 %	27	27
ui	0.0 %	0	559

(figures/eclemma.png)

coverage tool for Eclipse is EclEmma ([//www.eclemma.org/](http://www.eclemma.org/)) , shown on the right.

Lines that have been executed by the test suite are colored green, and lines not yet covered are red. If you saw this result from your coverage tool, your next step would be to come up with a test case that causes the body of the while loop to execute, and add it to your test suite so that the red lines become green.

Unit Testing and Stubs

A well-tested program will have tests for every individual module (where a module is a method or a class) that it contains. A test that tests an individual module, in isolation if possible, is called a **unit test**. Testing modules in isolation leads to much easier debugging. When a unit test for a module fails, you can be more confident that the bug is found in that module, rather than anywhere in the program.

The opposite of a unit test is an **integration test**, which tests a combination of modules, or even the entire program. If all you have are integration tests, then when a test fails, you have to hunt for the bug. It might be anywhere in the program. Integration tests are still important, because a program can fail at the connections between modules. For example, one module may be expecting different inputs than it's actually getting from another module. But if you have a thorough set of unit tests that give you confidence in the correctness of individual modules, then you'll have much less searching to do to find the bug.

Suppose you're building a web search engine. Two of your modules might be `getWebPage()` , which downloads web pages, and `extractWords()` , which splits a page into its component words:

```
/** @return the contents of the web page downloaded from url
 */
public static String getWebPage(URL url) {...}

/** @return the words in string s, in the order they appear,
 *         where a word is a contiguous sequence of
 *         non-whitespace and non-punctuation characters
 */
public static List<String> extractWords(String s) { ... }
```

These methods might be used by another module `makeIndex()` as part of the web crawler that makes the search engine's index:

```
/** @return an index mapping a word to the set of URLs
 *         containing that word, for all webpages in the input set
 */
public static Map<String, Set<URL>> makeIndex(Set<URL> urls) {
    ...
    for (URL url : urls) {
        String page = getWebPage(url);
        List<String> words = extractWords(page);
        ...
    }
    ...
}
```

In our test suite, we would want:

- unit tests just for `getWebPage()` that test it on various URLs
- unit tests just for `extractWords()` that test it on various strings
- unit tests for `makeIndex()` that test it on various sets of URLs

One mistake that programmers sometimes make is writing test cases for `extractWords()` in such a way that the test cases depend on `getWebPage()` to be correct. It's better to think about and test `extractWords()` in isolation, and partition it. Using test partitions that involve web page content might be reasonable, because that's how `extractWords()` is actually used in the program. But don't actually call `getWebPage()` from the test case, because `getWebPage()` may be buggy! Instead, store web page content as a literal string, and pass it directly to `extractWords()`. That way you're writing an isolated unit test, and if it fails, you can be more confident that the bug is in the module it's actually testing, `extractWords()`.

Note that the unit tests for `makeIndex()` can't easily be isolated in this way. When a test case calls `makeIndex()`, it is testing the correctness of not only the code inside `makeIndex()`, but also all the methods called by `makeIndex()`. If the test fails, the bug might be in any of those methods. That's why we want separate tests for `getWebPage()` and `extractWords()`, to increase our confidence in those modules individually and localize the problem to the `makeIndex()` code that connects them together.

Isolating a higher-level module like `makeIndex()` is possible if we write **stub** versions of the modules that it calls. For example, a stub for `getWebPage()` wouldn't access the internet at all, but instead would return mock web page content no matter what URL was passed to it. A stub for a class is often called a **mock object** ([//en.wikipedia.org/wiki/Mock_object](https://en.wikipedia.org/wiki/Mock_object)). Stubs are an important technique when building large systems, but we will generally not use them in 6.005.

Automated Testing and Regression Testing

Nothing makes tests easier to run, and more likely to be run, than complete automation. **Automated testing** means running the tests and checking their results automatically. A test driver should not be an interactive program that prompts you for inputs and prints out results for you to manually check. Instead, a test driver should invoke the module itself on fixed test cases and automatically check that the results are correct. The result of the test driver should be either “all tests OK” or “these tests failed: ...” A good testing framework, like JUnit, helps you build automated test suites.

Note that automated testing frameworks like JUnit make it easy to run the tests, but you still have to come up with good test cases yourself. *Automatic test generation* is a hard problem, still a subject of active computer science research.

Once you have test automation, it’s very important to rerun your tests when you modify your code. This prevents your program from *regressing* — introducing other bugs when you fix new bugs or add new features. Running all your tests after every change is called **regression testing**.

Whenever you find and fix a bug, take the input that elicited the bug and add it to your automated test suite as a test case. This kind of test case is called a *regression test*. This helps to populate your 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! Saving regression tests also protects against reversions that reintroduce the bug. The bug may be an easy error to make, since it happened once already.

This idea also leads to *test-first debugging*. When a bug arises, immediately write a test case for it that elicits it, and immediately add it to your test suite. Once you find and fix the bug, all your test cases will be passing, and you’ll be done with debugging and have a regression test for that bug.

In practice, these two ideas, automated testing and regression testing, are almost always used in combination.

Regression testing is only practical if the tests can be run often, automatically. Conversely, if you already have automated testing in place for your project, then you might as well use it to prevent regressions. So **automated regression testing** is a best-practice of modern software engineering.

Summary

In this reading, we saw these ideas:

- Test-first programming. Write tests before you write code.
- Partitioning and boundaries for choosing test cases systematically.
- White box testing and statement coverage for filling out a test suite.
- Unit-testing each module, in isolation as much as possible.
- Automated regression testing to keep bugs from coming back.

The topics of today’s reading connect to our three key properties of good software as follows:

- **Safe from bugs.** Testing is about finding bugs in your code, and test-first programming is about finding them as early as possible, immediately after you introduced them.
- **Easy to understand.** Testing doesn’t help with this as much as code review does.
- **Ready for change.** Readiness for change was considered by writing tests that only depend on behavior in the spec. We also talked about automated regression testing, which helps keep bugs from coming back when changes are made to code.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 4: Code Review

Code Review

Smelly Example #1

Don't Repeat Yourself

Comments Where Needed

Fail Fast

Avoid Magic Numbers

One Purpose For Each Variable

Smelly Example #2

Use Good Names

Use Whitespace to Help the Reader

Smelly Example #3

Don't Use Global Variables

Methods Should Return Results, not Print Them

Summary

Reading 4: Code Review

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives for Today's Class

In today's class, we will practice:

- code review: reading and discussing code written by somebody else
- general principles of good coding: things you can look for in every code review, regardless of programming language or program purpose

Code Review

Code review is careful, systematic study of source code by people who are not the original author of the code. It's analogous to proofreading a term paper.

Code review really has two purposes:

- **Improving the code.** Finding bugs, anticipating possible bugs, checking the clarity of the code, and checking for consistency with the project's style standards.
- **Improving the programmer.** Code review is an important way that programmers learn and teach each other, about new language features, changes in the design of the project or its coding standards, and new techniques. In open source projects, particularly, much conversation happens in the context of code reviews.

Code review is widely practiced in open source projects like Apache and Mozilla ([//blog.humphd.org/vocamus-1569/?p=1569](http://blog.humphd.org/vocamus-1569/?p=1569)) . Code review is also widely practiced in industry. At Google, you can't push any code into the main repository until another engineer has signed off on it in a code review.

In 6.005, we'll do code review on problem sets, as described in the Code Reviewing document ([..../general/code-review.html](#)) on the course website.

Style Standards

Most companies and large projects have coding style standards (for example, Google Java Style ([//google.github.io/styleguide/javaguide.html](http://google.github.io/styleguide/javaguide.html))) . These can get pretty detailed, even to the point of specifying whitespace (how deep to indent) and where curly braces and parentheses should go. These kinds of questions often lead to holy wars ([//www.outpost9.com/reference/jargon/jargon_23.html#TAG897](http://www.outpost9.com/reference/jargon/jargon_23.html#TAG897)) since they end up being a matter of taste and style.

For Java, there's a general style guide

([//www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html](http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html)) (unfortunately not updated for the latest versions of Java). Some of its advice gets very specific:

- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.

In 6.005, we have no official style guide of this sort. We're not going to tell you where to put your curly braces. That's a personal decision that each programmer should make. It's important to be self-consistent, however, and it's very important to follow the conventions of the project you're working on. If you're the programmer who reformats every module you touch to match your personal style, your teammates will hate you, and rightly so. Be a team player.

But there are some rules that are quite sensible and target our big three properties, in a stronger way than placing curly braces. The rest of this reading talks about some of these rules, at least the ones that are relevant at this point in the course, where we're mostly talking about writing basic Java. These are some things you should start to look for when you're code reviewing other students, and when you're looking at your own code for improvement. Don't consider it an exhaustive list of code style guidelines, however. Over the course of the semester, we'll talk about a lot more things — specifications, abstract data types with representation invariants, concurrency and thread safety — which will then become fodder for code review.

Smelly Example #1

Programmers often describe bad code as having a “bad smell” that needs to be removed. “Code hygiene” is another word for this. Let's start with some smelly code.

```

public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth;
}

```

The next few sections and exercises will pick out the particular smells in this code example.

Don't Repeat Yourself

Duplicated code is a risk to safety. If you have identical or very similar code in two places, then the fundamental risk is that there's a bug in both copies, and some maintainer fixes the bug in one place but not the other.

Avoid duplication like you'd avoid crossing the street without looking. Copy-and-paste is an enormously tempting programming tool, and you should feel a frisson of danger run down your spine every time you use it. The longer the block you're copying, the riskier it is.

Don't Repeat Yourself ([//en.wikipedia.org/wiki/Don't_repeat_yourself](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself)) , or DRY for short, has become a programmer's mantra.

The `dayOfYear` example is full of identical code. How would you DRY it out?

Comments Where Needed

A quick general word about commenting. Good software developers write comments in their code, and do it judiciously. Good comments should make the code easier to understand, safer from bugs (because important assumptions have been documented), and ready for change.

One kind of crucial comment is a specification, which appears above a method or above a class and documents the behavior of the method or class. In Java, this is conventionally written as a Javadoc comment, meaning that it starts with `/**` and includes `@`-syntax, like `@param` and `@return` for methods. Here's an example of a spec:

```
/**  
 * Compute the hailstone sequence.  
 * See http://en.wikipedia.org/wiki/Collatz\_conjecture#Statement\_of\_the\_problem  
 * @param n starting number of sequence; requires n > 0.  
 * @return the hailstone sequence starting at n and ending with 1.  
 *         For example, hailstone(3)=[3,10,5,16,8,4,2,1].  
 */  
public static List<Integer> hailstoneSequence(int n) {  
    ...  
}
```

Specifications document assumptions. We've already mentioned specs a few times, and there will be much more to say about them in a future reading.

Another crucial comment is one that specifies the provenance or source of a piece of code that was copied or adapted from elsewhere. This is vitally important for practicing software developers, and is required by the 6.005 collaboration policy ([..../general/collaboration.html](#)) when you adapt code you found on the web. Here is an example:

```
// read a web page into a string  
// see http://stackoverflow.com/questions/4328711/read-url-to-string-in-few-lines-of-java-code  
String mitHomepage = new Scanner(new URL("http://www.mit.edu").openStream(), "UTF-8").useDelimiter("\A").next();
```

One reason for documenting sources is to avoid violations of copyright. Small snippets of code on Stack Overflow are typically in the public domain, but code copied from other sources may be proprietary or covered by other kinds of open source licenses, which are more restrictive. Another reason for documenting sources is that the code can fall out of date; the Stack Overflow answer ([//stackoverflow.com/questions/4328711/read-url-to-string-in-few-lines-of-java-code](#)) from which this code came has evolved significantly in the years since it was first answered.

Some comments are bad and unnecessary. Direct transliterations of code into English, for example, do nothing to improve understanding, because you should assume that your reader at least knows Java:

```
while (n != 1) { // test whether n is 1  (don't write comments like this!)  
    ++i; // increment i  
    l.add(n); // add n to l  
}
```

But obscure code should get a comment:

```
sendMessage("as you wish"); // this basically says "I love you"
```

The `dayOfYear` code needs some comments — where would you put them? For example, where would you document whether `month` runs from 0 to 11 or from 1 to 12?

Fail Fast

Failing fast means that code should reveal its bugs as early as possible. The earlier a problem is observed (the closer to its cause), the easier it is to find and fix. As we saw in the first reading ([..01-static-checking/#static_checking_dynamic_checking_no_checking](#)), static checking fails faster than dynamic checking, and dynamic checking fails faster than producing a wrong answer that may corrupt subsequent computation.

The `day0fYear` function doesn't fail fast — if you pass it the arguments in the wrong order, it will quietly return the wrong answer. In fact, the way `day0fYear` is designed, it's highly likely that a non-American will pass the arguments in the wrong order! It needs more checking — either static checking or dynamic checking.

Avoid Magic Numbers

There are really only two constants that computer scientists recognize as valid in and of themselves: 0, 1, and maybe 2. (Okay, three constants.)

All other constants are called magic

([https://en.wikipedia.org/wiki/Magic_number_\(programming\)#Unnamed_numerical_constants](https://en.wikipedia.org/wiki/Magic_number_(programming)#Unnamed_numerical_constants)) because they appear as if out of thin air with no explanation.

One way to explain a number is with a comment, but a far better way is to declare the number as a named constant with a good, clear name.

`day0fYear` is full of magic numbers:

- The months 2, ..., 12 would be far more readable as `FEBRUARY`, ..., `DECEMBER`.
- The days-of-months 30, 31, 28 would be more readable (and eliminate duplicate code) if they were in a data structure like an array, list, or map, e.g. `MONTH_LENGTH[month]`.
- The mysterious numbers 59 and 90 are particularly pernicious examples of magic numbers. Not only are they uncommented and undocumented, they are actually the result of a *computation done by hand* by the programmer. Don't hardcode constants that you've computed by hand. Java is better at arithmetic than you are. Explicit computations like `31 + 28` make the provenance of these mysterious numbers much clearer. `MONTH_LENGTH[JANUARY] + MONTH_LENGTH[FEBRUARY]` would be clearer still.

One Purpose For Each Variable

In the `day0fYear` example, the parameter `day0fMonth` is reused to compute a very different value — the return value of the function, which is not the day of the month.

Don't reuse parameters, and don't reuse variables. Variables are not a scarce resource in programming. Introduce them freely, give them good names, and just stop using them when you stop needing them. You will confuse your reader if a variable that used to mean one thing suddenly starts meaning something different a few lines down.

Not only is this an ease-of-understanding question, but it's also a safety-from-bugs and ready-for-change question.

Method parameters, in particular, should generally be left unmodified. (This is important for being ready-for-change — in the future, some other part of the method may want to know what the original parameters of the method were, so you shouldn't blow them away while you're computing.) It's a good idea to use `final` for method parameters, and as many other variables as you can. The `final` keyword says that the variable should never be reassigned, and the Java compiler will check it statically. For example:

```
public static int day0fYear(final int month, final int day0fMonth, final int year) {  
    ...  
}
```

Smelly Example #2

There was a latent bug in `day0fYear`. It didn't handle leap years at all. As part of fixing that, suppose we write a leap-year method.

```
public static boolean leap(int y) {
    String tmp = String.valueOf(y);
    if (tmp.charAt(2) == '1' || tmp.charAt(2) == '3' || tmp.charAt(2) == 5 || tmp.charAt(2) == '7' || tmp.charAt(2) == '9') {
        if (tmp.charAt(3)=='2'||tmp.charAt(3)=='6') return true; /*R1*/
        else
            return false; /*R2*/
    }else{
        if (tmp.charAt(2) == '0' && tmp.charAt(3) == '0') {
            return false; /*R3*/
        }
        if (tmp.charAt(3)=='0'||tmp.charAt(3)=='4'||tmp.charAt(3)=='8')return true; /*R4*/
    }
    return false; /*R5*/
}
```

What are the bugs hidden in this code? And what style problems that we've already talked about?

Use Good Names

Good method and variable names are long and self-descriptive. Comments can often be avoided entirely by making the code itself more readable, with better names that describe the methods and variables.

For example, you can rewrite

```
int tmp = 86400; // tmp is the number of seconds in a day (don't do this!)
```

as:

```
int secondsPerDay = 86400;
```

In general, variable names like `tmp`, `temp`, and `data` are awful, symptoms of extreme programmer laziness. Every local variable is temporary, and every variable is data, so those names are generally meaningless. Better to use a longer, more descriptive name, so that your code reads clearly all by itself.

Follow the lexical naming conventions of the language. In Python, classes are typically Capitalized, variables are lowercase, and words_are_separated_by_underscores. In Java:

- `methodsAreNamedWithCamelCaseLikeThis`
- `variablesAreAlsoCamelCase`
- `CONSTANTS_ARE_IN_ALL_CAPS_WITH_UNDERSCORES`
- `ClassesAreCapitalized`
- `packages.are.lowercase.and.separated.by.dots`

Method names are usually verb phrases, like `getDate` or `isUpperCase`, while variable and class names are usually noun phrases. Choose short words, and be concise, but avoid abbreviations. For example, `message` is clearer than `msg`, and `word` is so much better than `wd`. Keep in mind that many of your teammates in class and in the real world will not be native English speakers, and abbreviations can be even harder for non-native speakers.

The `leap` method has bad names: the method name itself, and the local variable name. What would you call them instead?

Use Whitespace to Help the Reader

Use consistent indentation. The `leap` example is bad at this. The `day0fYear` example is much better. In fact, `day0fYear` nicely lines up all the numbers into columns, making them easy for a human reader to compare and check. That's a great use of whitespace.

Put spaces within code lines to make them easy to read. The `leap` example has some lines that are packed together — put in some spaces.

Never use tab characters for indentation, only space characters. Note that we say *characters*, not keys. We're not saying you should never press the Tab key, only that your editor should never put a tab character into your source file in response to your pressing the Tab key. The reason for this rule is that different tools treat tab characters differently — sometimes expanding them to 4 spaces, sometimes to 2 spaces, sometimes to 8. If you run “git diff” on the command line, or if you view your source code in a different editor, then the indentation may be completely screwed up. Just use spaces. Always set your programming editor to insert space characters when you press the Tab key.

Smelly Example #3

Here's a third example of smelly code that will illustrate the remaining points of this reading.

```
public static int LONG_WORD_LENGTH = 5;
public static String longestWord;

public static void countLongWords(List<String> words) {
    int n = 0;
    longestWord = "";
    for (String word: words) {
        if (word.length() > LONG_WORD_LENGTH) ++n;
        if (word.length() > longestWord.length()) longestWord = word;
    }
    System.out.println(n);
}
```

Don't Use Global Variables

Avoid global variables. Let's break down what we mean by *global variable*. A global variable is:

- a *variable*, a name whose meaning can be changed
- that is *global*, accessible and changeable from anywhere in the program.

Why Global Variables Are Bad ([//c2.com/cgi/wiki?GlobalVariablesAreBad](http://c2.com/cgi/wiki?GlobalVariablesAreBad)) has a good list of the dangers of global variables.

In Java, a global variable is declared `public static`. The `public` modifier makes it accessible anywhere, and `static` means there is a single instance of the variable.

In general, change global variables into parameters and return values, or put them inside objects that you're calling methods on. We'll see many techniques for doing that in future readings.

Methods Should Return Results, not Print Them

`countLongWords` isn't ready for change. It sends some of its result to the console, `System.out`. That means that if you want to use it in another context — where the number is needed for some other purpose, like computation rather than human eyes — it would have to be rewritten.

In general, only the highest-level parts of a program should interact with the human user or the console. Lower-level parts should take their input as parameters and return their output as results. The sole exception here is debugging output, which can of course be printed to the console. But that kind of output shouldn't be a part of your design, only a part of how you debug your design.

Summary

Code review is a widely-used technique for improving software quality by human inspection. Code review can detect many kinds of problems in code, but as a starter, this reading talked about these general principles of good code:

- Don't Repeat Yourself (DRY)
- Comments where needed
- Fail fast
- Avoid magic numbers
- One purpose for each variable
- Use good names
- No global variables
- Return results, don't print them
- Use whitespace for readability

The topics of today's reading connect to our three key properties of good software as follows:

- **Safe from bugs.** In general, code review uses human reviewers to find bugs. DRY code lets you fix a bug in only one place, without fear that it has propagated elsewhere. Commenting your assumptions clearly makes it less likely that another programmer will introduce a bug. The Fail Fast principle detects bugs as early as possible. Avoiding global variables makes it easier to localize bugs related to variable values, since non-global variables can be changed in only limited places in the code.
- **Easy to understand.** Code review is really the only way to find obscure or confusing code, because other people are reading it and trying to understand it. Using judicious comments, avoiding magic numbers, keeping one purpose for each variable, using good names, and using whitespace well can all improve the understandability of code.
- **Ready for change.** Code review helps here when it's done by experienced software developers who can anticipate what might change and suggest ways to guard against it. DRY code is more ready for change, because a change only needs to be made in one place. Returning results instead of printing them makes it easier to adapt the code to a new purpose.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 5: Version Control

Introduction

Inventing version control

Git

Copy an object graph with git clone

What else is in the object graph?

Add to the object graph with git commit

Send & receive object graphs with git push & git pull

Merging

Why do commits look like diffs?

Version control and the big three

Reading 5: Version Control

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

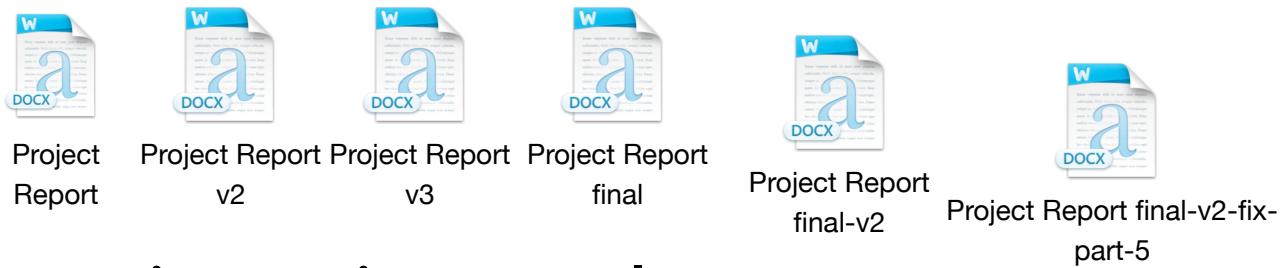
- Know what version control is and why we use it
- Understand how Git stores version history as a graph
- Practice reading, creating, and using version history

Introduction

Version control systems ([//en.wikipedia.org/wiki/Revision_control](https://en.wikipedia.org/wiki/Revision_control)) are essential tools of the software engineering world. More or less every project — serious or hobby, open source or proprietary — uses version control. Without version control, coordinating a team of programmers all editing the same project's code will reach pull-out-your-hair levels of aggravation.

Version control systems you've already used

- Dropbox
- Undo/redo buffer ([//en.wikipedia.org/wiki/Undo](https://en.wikipedia.org/wiki/Undo))
- Keeping multiple copies of files with version numbers

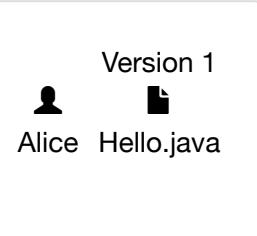


Inventing version control

Suppose Alice ([//en.wikipedia.org/wiki/Alice_and_Bob](https://en.wikipedia.org/wiki/Alice_and_Bob)) is working on a problem set by herself.

She starts with one file `Hello.java` in her pset, which she works on for several days.

At the last minute before she needs to hand in her pset to be graded, she realizes she has made a change that breaks everything. If only she could go back in time and retrieve a past version!



A simple discipline of saving backup files would get the job done.

Alice uses her judgment to decide when she has reached some milestone that justifies saving the code. She saves the versions of `Hello.java` as `Hello.1.java`, `Hello.2.java`, and `Hello.java`. She follows the convention that the most recent version is just `Hello.java` to avoid confusing Eclipse. We will call the most recent version the *head*.

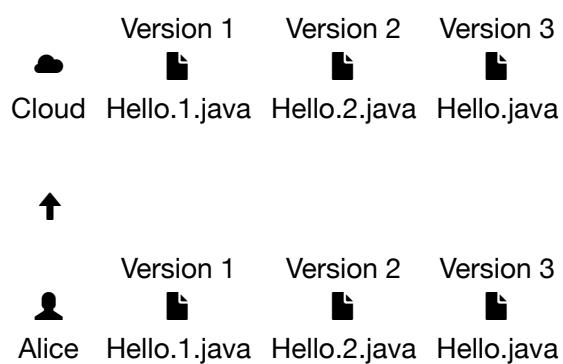


Now when Alice realizes that version 3 is fatally flawed, she can just copy version 2 back into the location for her current code. Disaster averted! But what if version 3 included some changes that were good and some that were bad? Alice can compare the files manually to find the changes, and sort them into good and bad changes. Then she can copy the good changes into version 2.

This is a lot of work, and it's easy for the human eye to miss changes. Luckily, there are standard software tools for comparing text; in the UNIX world, one such tool is `diff` ([/en.wikipedia.org/wiki/Diff](https://en.wikipedia.org/wiki/Diff)). A better version control system will make diffs easy to generate.

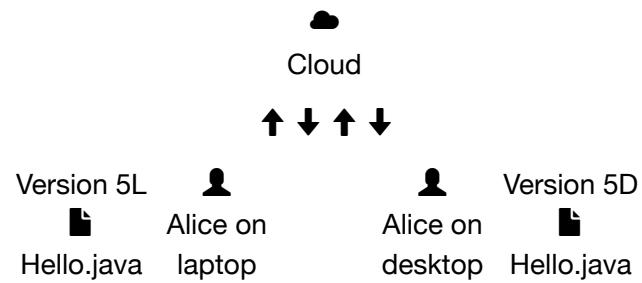
Alice also wants to be prepared in case her laptop gets run over by a bus, so she saves a backup of her work in the cloud, uploading the contents of her working directory whenever she's satisfied with its contents.

If her laptop is kicked into the Charles, Alice can retrieve the backup and resume work on the pset on a fresh machine, retaining the ability to time-travel back to old versions at will.



Furthermore, she can develop her pset on multiple machines, using the cloud provider as a common interchange point. Alice makes some changes on her laptop and uploads them to the cloud. Then she downloads onto her desktop machine at home, does some more work, and uploads the improved code (complete with old file versions) back to the cloud.

If Alice isn't careful, though, she can run into trouble with this approach. Imagine that she starts editing `Hello.java` to create "version 5" on her laptop. Then she gets distracted and forgets about her changes. Later, she starts working on a new "version 5" on her desktop machine, including *different* improvements. We'll call these versions "5L" and "5D," for "laptop" and "desktop."



When it comes time to upload changes to the cloud, there is an opportunity for a mishap! Alice might copy all her local files into the cloud, causing it to contain version 5D only. Later Alice syncs from the cloud to her laptop, potentially overwriting version 5L, losing the worthwhile changes. What Alice really wants here is a *merge*, to create a new version based on the two version 5's.

At this point, considering just the scenario of one programmer working alone, we already have a list of operations that should be supported by a version control scheme:

- *reverting* to a past version
- *comparing* two different versions
- *pushing* full version history to another location
- *pulling* history back from that location
- *merging* versions that are offshoots of the same earlier version

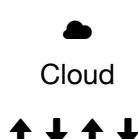
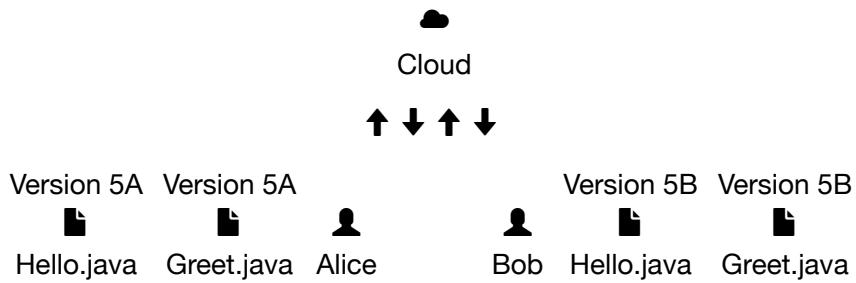
Multiple developers

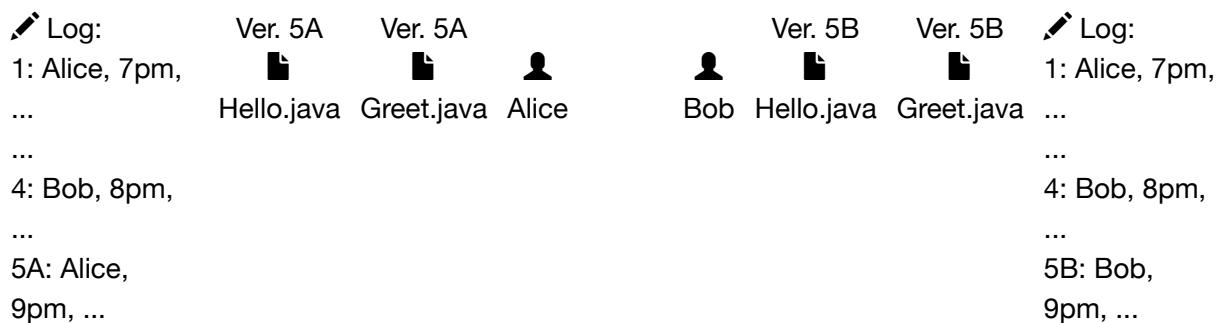
Now let's add into the picture Bob, another developer. The picture isn't too different from what we were just thinking about.

Alice and Bob here are like the two Alices working on different computers. They no longer share a brain, which makes it even more important to follow a strict discipline in pushing to and pulling from the shared cloud server. The two programmers must coordinate on a scheme for coming up with

version numbers. Ideally, the scheme allows us to assign clear names to *whole sets of files*, not just individual files. (Files depend on other files, so thinking about them in isolation allows inconsistencies.)

Merely uploading new source files is not a very good way to communicate to others the high-level idea of a set of changes. So let's add a log that records for each version *who* wrote it, *when* it was finalized, and *what* the changes were, in the form of a short human-authored message.





Pushing another version now gets a bit more complicated, as we need to merge the logs. This is easier to do than for Java files, since logs have a simpler structure – but without tool support, Alice and Bob will need to do it manually! We also want to enforce consistency between the logs and the actual sets of available files: for each log entry, it should be easy to extract the complete set of files that were current at the time the entry was made.

But with logs, all sorts of useful operations are enabled. We can look at the log for just a particular file: a view of the log restricted to those changes that involved modifying some file. We can also use the log to figure out which change contributed each line of code, or, even better, which person contributed each line, so we know who to complain to when the code doesn't work. This sort of operation would be tedious to do manually; the automated operation in version control systems is called *annotate* (or, unfortunately, *blame*).

Multiple branches

It sometimes makes sense for a subset of the developers to go off and work on a *branch*, a parallel code universe for, say, experimenting with a new feature. The other developers don't want to pull in the new feature until it is done, even if several coordinated versions are created in the meantime. Even a single developer can find it useful to create a branch, for the same reasons that Alice was originally using the cloud server despite working alone.

In general, it will be useful to have many shared places for exchanging project state. There may be multiple branch locations at once, each shared by several programmers. With the right set-up, any programmer can pull from or push to any location, creating serious flexibility in cooperation patterns.

The shocking conclusion

Of course, it turns out we haven't invented anything here: Git ([//git-scm.com](http://git-scm.com)) does all these things for you, and so do many other version control systems.

Distributed vs. centralized

Traditional *centralized* version control systems like CVS and Subversion ([//subversion.apache.org/](http://subversion.apache.org/)) do a subset of the things we've imagined above. They support a collaboration graph – who's sharing what changes with whom – with one master server and copies that only communicate with the master.

In a centralized system, everyone must share their work to and from the master repository. Changes are safely stored *in version control* if they are *in the master repository*, because that's the only repository.

Dan Carol



Cloud



Alice Bob

Dan Carol



Cloud



Alice Bob

In contrast, *distributed* version control systems like Git ([//git-scm.com](http://git-scm.com)) and Mercurial (<https://www.mercurial-scm.org/>) allow all sorts of different collaboration graphs, where teams and subsets of teams can experiment easily with alternate versions of code and history, merging versions together as they are determined to be good ideas.

In a distributed system, all repositories are created equal, and it's up to users to assign them different roles. Different users might share their work to and from different repos, and the team must decide what it means for a change to be *in version control*. If the change is stored in just a single programmer's repo, do they still need to share it with a designated collaborator or specific server before the rest of the team considers it official?

Version control terminology

- **Repository** : a local or remote store of the versions in our project
- **Working copy** : a local, editable copy of our project that we can work on
- **File** : a single file in our project
- **Version or revision** : a record of the contents of our project at a point in time
- **Change or diff** : the difference between two versions
- **Head** : the current version

Features of a version control system

- **Reliable** : keep versions around for as long as we need them; allow backups
- **Multiple files** : track versions of a project, not single files
- **Meaningful versions** : what were the changes, why where they made?
- **Revert** : restore old versions, in whole or in part
- **Compare versions**
- **Review history** : for the whole project or individual files
- **Not just for code** : prose, images, ...

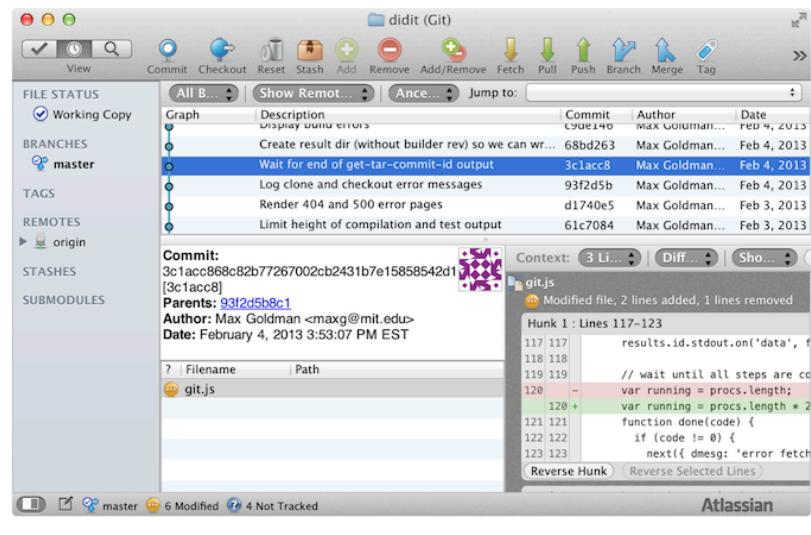
It should **allow multiple people to work together** :

- **Merge** : combine versions that diverged from a common previous version
- **Track responsibility** : who made that change, who touched that line of code?
- **Work in parallel** : allow one programmer to work on their own for a while (without giving up version control)
- **Work-in-progress** : allow multiple programmers to share unfinished work (without disrupting others, without giving up version control)

Git

The version control system we'll use in 6.005 is Git ([//git-scm.com](http://git-scm.com)) . It's powerful and worth learning. But Git's user interface can be terribly frustrating. What is Git's user interface?

- In 6.005, we will use Git on the command line. The command line is a fact of life, ubiquitous because it is so powerful.
- The command line can make it very difficult to see what is going on in your repositories. You may find SourceTree ([//www.sourcetreeapp.com](http://www.sourcetreeapp.com)) (shown on the right) for Mac & Windows useful. On any platform, gitk ([//git-scm.com/docs/gitk](http://git-scm.com/docs/gitk)) can give you a basic Git GUI. Ask Google for other suggestions.



An important note about tools for Git:

- Eclipse has built-in support for Git. If you follow the problem set instructions ([..../psets/ps0/](#)), Eclipse will know your project is in Git and will show you helpful icons. We do not recommend using the Eclipse Git UI to make changes, commit, etc., and course staff may not be able to help you with problems.
- GitHub ([//github.com/](http://github.com/)) makes desktop apps for Mac and Windows. Because the GitHub app changes how some Git operations work, if you use the GitHub app, course staff will not be able to help you.

Getting started with Git

On the Git ([//git-scm.com](http://git-scm.com)) website, you can find two particularly useful resources:

- *Pro Git* ([//git-scm.com/book](http://git-scm.com/book)) documents everything you might need to know about Git.
- The Git command reference ([//git-scm.com/docs](http://git-scm.com/docs)) can help with the syntax of Git commands.

You've already completed **PS0** ([..../psets/ps0/#clone](#)) and the **Getting Started intro to Git** ([..../getting-started/#git](#)) .

The Git object graph

Read: **Pro Git 1.3: Git Basics** ([//git-scm.com/book/en/v2/Getting-Started-Git-Basics](http://git-scm.com/book/en/v2/Getting-Started-Git-Basics))

That reading introduces the three pieces of a Git repo: `.git` directory, working directory, and staging area.

All of the operations we do with Git — clone, add, commit, push, log, merge, ... — are operations on a graph data structure that stores all of the versions of files in our project, and all the log entries describing those changes. The **Git object graph** is stored in the `.git` directory of your local repository. Another copy of the graph, e.g. for PS0, is on Athena in:

`/mit/6.005/git/sp16/psets/ps0/[your username].git`

Copy an object graph with `git clone`

How do you get the object graph from Athena to your local machine in order to start working on the problem set? `git clone` copies the graph.

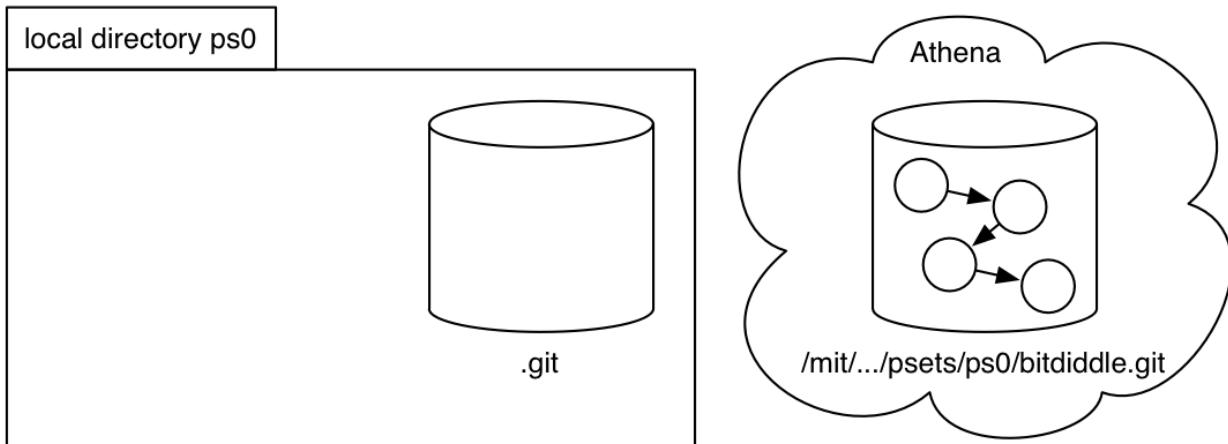
Suppose your username is `bitdiddle` :

```
git clone ssh://.../psets/ps0/bitdiddle.git ps0
```

Hover or tap on each step to update the diagram below:

1. Create an empty local directory `ps0` , and `ps0/.git` .
2. Copy the object graph from `ssh://.../psets/ps0/bitdiddle.git` into `ps0/.git` .
3. **Check out** the current version of the `master` branch .

Diagram for highlighted step:



We still haven't explained what's in the object graph. But before we do that, let's understand step 3 of `git clone` : check out the current version of the `master` branch.

The object graph is stored on disk in a convenient and efficient structure for performing Git operations, but not in a format we can easily use. In Alice's invented version control scheme , the current version of `Hello.java` was just called `Hello.java` because she needed to be able to edit it normally. In Git, we obtain normal copies of our files by *checking them out* from the object graph. These are the files we see and edit in Eclipse.

We also decided above that it might be useful to support multiple *branches* in the version history . Multiple branches are essential for large teams working on long-term projects. To keep things simple in 6.005, we will not use branches and we don't recommend that you create any. Every Git repo comes with a default branch called `master` , and all of our work will be on the `master` branch.

So step 2 of `git clone` gets us an object graph, and step 3 gets us a **working directory** full of files we can edit, starting from the current version of the project.

Let's finally dive into that object graph!

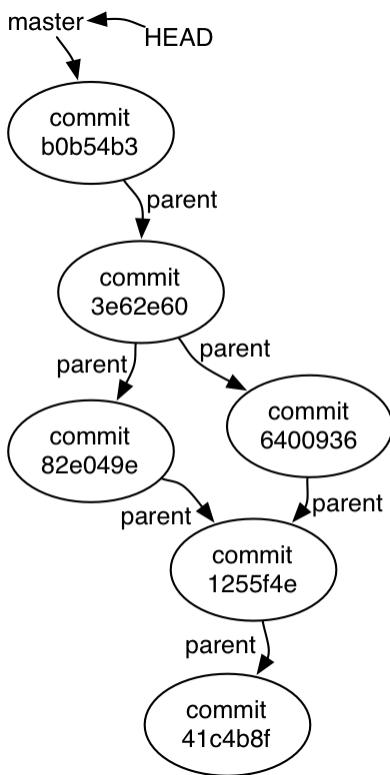
Clone an example repo: <https://github.com/mit6005/sp16-ex05-hello-git.git>

Using commands from **Getting Started** ([../../getting-started/#git](#)) or **Pro Git 2.3: Viewing the Commit History** ([//git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History](#)) , or by using a tool like SourceTree, explain the history of this little project to yourself.

Here's the output of `git log` (./../getting-started/#config-git) for this example repository:

```
* b0b54b3 (HEAD, origin/master, origin/HEAD, master) Greeting in Java
* 3e62e60 Merge
| \
| * 6400936 Greeting in Scheme
* | 82e049e Greeting in Ruby
| /
* 1255f4e Change the greeting
* 41c4b8f Initial commit
```

The history of a Git project is a **directed acyclic graph** ([//en.wikipedia.org/wiki/Directed_acyclic_graph](https://en.wikipedia.org/wiki/Directed_acyclic_graph)) (DAG). The history graph is the backbone of the full object graph stored in `.git`, so let's focus on it for a minute.



Each node in the history graph is a **commit** a.k.a. **version** a.k.a. **revision** of the project: a complete snapshot of all the files in the project at that point in time. You may recall from our earlier reading (./../getting-started/#getting_the_history_of_the_repository) that each commit is identified by a unique ID, displayed as a hexadecimal number.

Except for the initial commit, each commit has a pointer to its **parent** commit. For example, commit `1255f4e` has parent `41c4b8f` : this means `41c4b8f` happened first, then `1255f4e`.

Some commits have the same parent: they are versions that diverged from a common previous version. And some commits have two parents: they are versions that tie divergent histories back together.

A branch — remember `master` will be our only branch for now — is just a name that points to a commit.

Finally, HEAD points to our current commit — almost. We also need to remember which branch we're working on. So HEAD points to the current branch, which points to the current commit.

Check your understanding...

What else is in the object graph?

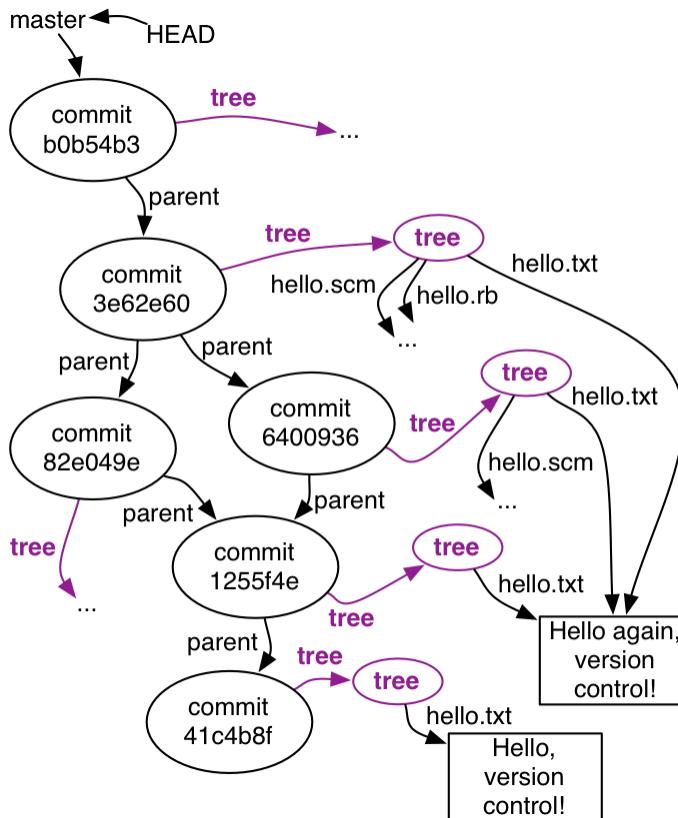
The history graph is the backbone of the full object graph. What else is in there?

Each commit is a snapshot of our entire project, which Git represents with a **tree** node. For a project of any reasonable size, most of the files *won't* change in any given revision. Storing redundant copies of the files would be wasteful, so Git doesn't do that.

Instead, the Git object graph stores each version of an individual file *once*, and allows multiple commits to *share* that one copy. To the left is a more complete rendering of the Git object graph for our example.

Keep this picture in the back of your mind, because it's a wonderful example of the sharing enabled by *immutable data types*, which we're going to discuss a few classes from now.

Each commit also has log data — who, when, short log message, etc. — not shown in the diagram.



Add to the object graph with `git commit`

How do we add new commits to the history graph? `git commit` creates a new commit.

In some alternate universe, `git commit` might create a new commit based on the current contents of your working directory. So if you edited `Hello.java` and then did `git commit`, the snapshot would include your changes.

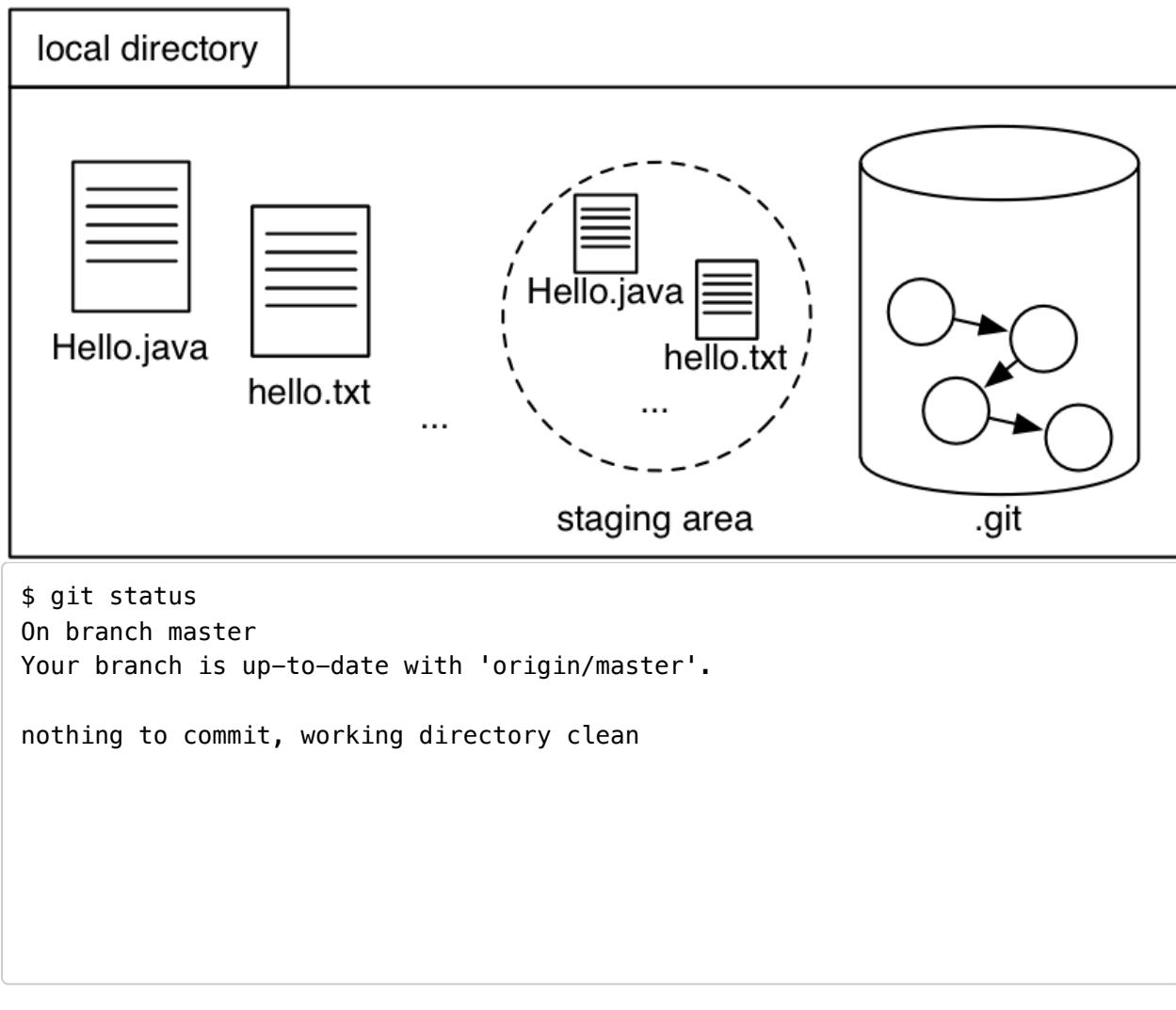
We're not in that universe; in our universe, Git uses that third and final piece of the repository: the **staging area** (a.k.a. the **index**, which is only a useful name to know because sometimes it shows up in documentation).

The staging area is like a proto-commit, a commit-in-progress. Here's how we use the staging area and `git add` to build up a new snapshot, which we then cast in stone using `git commit`:

```
Modify hello.txt , git add hello.txt , git commit
```

Hover or tap on each step to update the diagram, and to see the output of `git status` at each step:

1. If we haven't made any changes yet, then the working directory, staging area, and HEAD commit are all identical.
2. Make a change to a file. For example, let's edit `hello.txt`. Other changes might be creating a new file, or deleting a file.
3. **Stage** those changes using `git add`.
4. Create a new commit out of all the staged changes using `git commit`.



```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean
```

Use `git status` frequently to keep track of whether you have no changes, unstaged changes, or staged changes; and whether you have new commits in your local repository that haven't been pushed.

Sequences, trees, and graphs

When you're working independently, on a single machine, the DAG of your version history will usually look like a sequence: commit 1 is the parent of commit 2 is the parent of commit 3...

There are three programmers involved in the history of our example repository. Two of them – Alyssa and Ben – made changes “at the same time.” In this case, “at the same time” doesn’t mean precisely contemporaneous. Instead, it means they made two different *new* versions based on the same *previous* version, just as Alice made version 5L and 5D on her laptop and desktop .

When multiple commits share the same parent commit, our history DAG changes from a sequence to a tree: it branches apart. Notice that a branch in the history of the project doesn’t require anyone to create a new Git branch, merely that we start from the same commit and work in parallel on different copies of the repository:

```
:
* commit 82e049e248c63289b8a935ce71b130a74dc04152
| Author: Ben Bitdiddle <ben.bitdiddle@example.com>
| Greeting in Ruby
|
| * commit 64009369c5ab93492931ad07962ee81bda921ded
|/ Author: Alyssa P. Hacker <alyssa.p.hacker@example.com>
| Greeting in Scheme
|
* commit 1255f4e4a5836501c022deb337fd3f8800b02e4
| Author: Max Goldman <maxg@mit.edu>
| Change the greeting
:
```

Finally, the history DAG changes from tree- to graph-shaped when the branching changes are merged together:

```
:
* commit 3e62e60a7b4a0c262cd8eb4308ac3e5a1e94d839
|\ Author: Max Goldman <maxg@mit.edu>
| | Merge
| |
* | commit 82e049e248c63289b8a935ce71b130a74dc04152
| | Author: Ben Bitdiddle <ben.bitdiddle@example.com>
| | Greeting in Ruby
| |
| * commit 64009369c5ab93492931ad07962ee81bda921ded
|/ Author: Alyssa P. Hacker <alyssa.p.hacker@example.com>
| Greeting in Scheme
|
* commit 1255f4e4a5836501c022deb337fd3f8800b02e4
| Author: Max Goldman <maxg@mit.edu>
| Change the greeting
:
```

How is it that changes are merged together? First we'll need to understand how history is shared between different users and repositories.

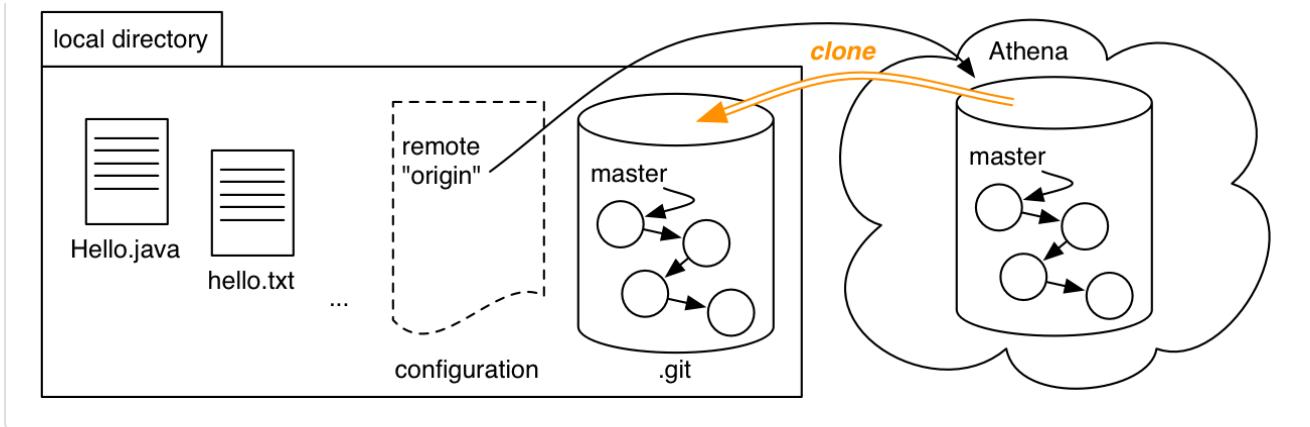
Send & receive object graphs with `git push` & `git pull`

We can send new commits to a remote repository using `git push` :

```
git push origin master
```

Hover or tap on each step to update the diagram:

1. When we clone a repository, we obtain a copy of the history graph.
Git remembers where we cloned from as a **remote repository** called `origin` .
2. Using `git commit` , we add new commits to the local history on the `master` branch.
3. To send those changes back to the `origin` remote, use `git push origin master` .



And we receive new commits using `git pull`. Note that `git pull`, in addition to fetching new parts of the object graph, also updates the working copy by checking out the latest version (just like `git clone` checked out a working copy to start with).

Merging

Now, let's examine what happens when changes occur in parallel:

Create and commit `hello.scm` and `hello.rb` in parallel

Hover or tap on each step to update the diagram:

1. Both Alyssa and Ben **clone** the repository with two commits (`41c4b8f` and `1255f4e`).
2. Alyssa creates `hello.scm` and **commits** her change as `6400936` .
3. At the same time, Ben creates `hello.rb` and **commits** his change as `82e049e` .

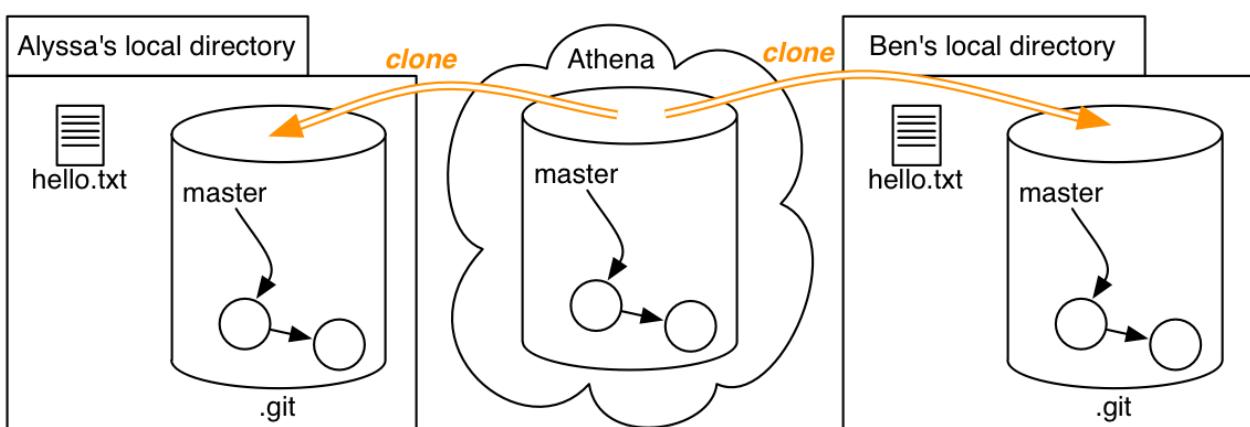
At this point, both of their changes only exist in their local repositories. In each repo, `master` now points to a different commit.

4. Let's suppose Alyssa is the first to **push** her change up to Athena.
5. What happens if Ben tries to push now? The push will be rejected: if the server updates `master` to point to Ben's commit, Alyssa's commit will disappear from the project history!
6. Ben must **merge** his changes with Alyssa's.

To perform the merge, he **pulls** her commit from Athena, which does two things:

- (a) Downloads new commits into Ben's repository's object graph

7. (b) Merges Ben's history with Alyssa's, creating a new commit (`3e62e60`) that joins together the disparate histories. This commit is a snapshot like any other: a snapshot of the repository with both of their changes applied.
8. Now Ben can `git push`, because no history will go missing when he does.
9. And Alyssa can `git pull` to obtain Ben's work.



In this example, Git was able to merge Alyssa's and Ben's changes automatically, because they each modified different files. If both of them had edited the *same parts of the same files*, Git would report a **merge conflict**. Ben would have to manually weave their changes together before committing the merge. All of this is discussed in the Getting Started section on merges, merging, and merge conflicts (./../getting-started/#merges).

Why do commits look like diffs?

We've defined a commit as a snapshot of our entire project, but if you ask Git, it doesn't seem to see things that way:

```
$ git show 1255f4e
commit 1255f4e4a5836501c022deb337fd3f8800b02e4
Author: Max Goldman <maxg@mit.edu>
Date:   Mon Sep 14 14:58:40 2015 -0400

    Change the greeting

diff --git a/hello.txt b/hello.txt
index c1106ab..3462165 100644
--- a/hello.txt
+++ b/hello.txt
@@ -1 +1 @@
-Hello, version control!
+Hello again, version control!
```

Git is assuming that most of our project does not change in any given commit, so showing only the differences will be more useful. Almost all the time, that's true.

But we can ask Git to show us what was in the repo at a particular commit:

```
$ git show 3e62e60:
tree 3e62e60:

hello.rb
hello.scm
hello.txt
```

Yes, the addition of a : completely changes the meaning of that command.

We can also see what was in a particular file in that commit:

```
$ git show 3e62e60:hello.scm
(display "Hello, version control!")
```

This is one of the simplest ways you can use Git to recover from a disaster: ask it to `git show` you the contents of a now-broken file at some earlier version when the file was OK.

We'll practice some disaster recovery commands in class.

Version control and the big three

How does version control relate to the three big ideas of 6.005?

Safe from bugs find when and where something broke
look for other, similar mistakes
gain confidence that code hasn't changed accidentally

Easy to understand why was a change made?
what else was changed at the same time?
who can I ask about this code?

Ready for change all about managing and organizing changes
accept and integrate changes from other developers
isolate speculative work on branches

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 6: Specifications

Introduction

Part 1: Specifications

Part 2: Exceptions

Summary

Reading 6: Specifications

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- Understand preconditions and postconditions in method specifications, and be able to write correct specifications
- Be able to write tests against a specification
- Know the difference between checked and unchecked exceptions in Java
- Understand how to use exceptions for special results

Introduction

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 precondition, the client has responsibilities too.

In this reading we'll look at the role played by specifications of methods. We'll discuss what preconditions and postconditions are, and what they mean for the implementor and the client of a method. We'll also talk about how to use exceptions, an important language feature found in Java, Python, and many other modern languages, which allows us to make a method's interface safer from bugs and easier to understand.

Part 1: Specifications (specs/)

Part 2: Exceptions (exceptions/)

Summary

Before we wrap up, check your understanding with one last example:

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.

Let's review how specifications help with the main goals of this course:

- **Safe from bugs** . A good specification clearly documents the mutual assumptions that a client and implementor are relying on. Bugs often come from disagreements at the interfaces, and the presence of a specification reduces that. Using machine-checked language features in your spec, like static typing and exceptions rather than just a human-readable comment, can reduce bugs still more.
- **Easy to understand** . A short, simple spec is easier to understand than the implementation itself, and saves other people from having to read the code.
- **Ready for change** . Specs establish contracts between different parts of your code, allowing those parts to change independently as long as they continue to satisfy the requirements of the contract.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 7: Designing Specifications

Introduction

Deterministic vs. underdetermined specs

Declarative vs. operational specs

Stronger vs. weaker specs

Diagramming specifications

Designing good specifications

Precondition or postcondition?

About access control

About static vs. instance methods

Summary

Reading 7: Designing Specifications

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- Understand underdetermined specs, and be able to identify and assess nondeterminism
- Understand declarative vs. operational specs, and be able to write declarative specs
- Understand strength of preconditions, postconditions, and specs; and be able to compare spec strength
- Be able to write coherent, useful specifications of appropriate strength

Introduction

In this reading we'll look at different specs for similar behaviors, and talk about the tradeoffs between them. We'll look at three dimensions for comparing specs:

- How **deterministic** it is. Does the spec define only a single possible output for a given input, or allow the implementor to choose from a set of legal outputs?
- How **declarative** it is. Does the spec just characterize *what* the output should be, or does it explicitly say *how* to compute the output?
- How **strong** it is. Does the spec have a small set of legal implementations, or a large set?

Not all specifications we might choose for a module are equally useful, and we'll explore what makes some specifications better than others.

Deterministic vs. underdetermined specs

Recall the two example implementations of `find` we began with in the previous reading (./06-specifications) :

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

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

The subscripts `First` and `Last` are not actual Java syntax. We're using them here to distinguish the two implementations for the sake of discussion. In the actual code, both implementations should be Java methods called `find`.

Here is one possible specification of `find` :

```
static int findExactlyOne(int[] arr, int val)
    requires: val occurs exactly once in arr
    effects: returns index i such that arr[i] = val
```

This specification is **deterministic** : when presented with a state satisfying the precondition, the outcome is completely determined. Only one return value and one final state is possible. There are no valid inputs for which there is more than one valid output.

Both `findFirst` and `findLast` satisfy the specification, so if this is the specification on which the clients relied, the two implementations are equivalent and substitutable for one another.

Here is a slightly different specification:

```
static int findOneOrMore,AnyIndex(int[] arr, int val)
    requires: val occurs in arr
    effects: returns index i such that arr[i] = val
```

This specification is not deterministic. It 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 allows multiple valid outputs for the same input.

Note that this is different from *nondeterministic* in the usual sense of that word. Nondeterministic code sometimes behaves one way and sometimes another, even if called in the same program with the same inputs. This can happen, for example, when the code's behavior depends on a random number, or when

it depends on the timing of concurrent processes. But a specification which is not deterministic doesn't have to have a nondeterministic implementation. It can be satisfied by a fully deterministic implementation.

To avoid the confusion, we'll refer to specifications that are not deterministic as **underdetermined**.

This underdetermined `find` spec is again satisfied by both `find First` and `find Last`, each resolving the underdeterminedness in its own (fully deterministic) way. A client of `find OneOrMore,AnyIndex` spec can't rely on which index will be returned if `val` appears more than once. The spec would be satisfied by a nondeterministic implementation, too — for example, one that tosses a coin to decide whether to start searching from the beginning or the end of the array. But in almost all cases we'll encounter, underdeterminism in specifications offers a choice that is made by the implementor at implementation time. An underdetermined spec is typically implemented by a fully-deterministic implementation.

Declarative vs. operational specs

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 inadvertently expose implementation details 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.

One reason programmers sometimes lapse into operational specifications is because they're using the spec comment to explain the implementation for a maintainer. Don't do that. When it's necessary, use comments within the body of the method, not in the spec comment.

For a given specification, there may be many ways to express it declaratively:

```
static boolean startsWith(String str, String prefix)
  effects: returns true if and only if there exists String suffix
            such that prefix + suffix == str
```

```
static boolean startsWith(String str, String prefix)
  effects: returns true if and only if there exists integer i
            such that str.substring(0, i) == prefix
```

```
static boolean startsWith(String str, String prefix)
  effects: returns true if the first prefix.length() characters of str
            are the characters of prefix, false otherwise
```

It's up to us to choose the clearest specification for clients and maintainers of the code.

Stronger vs. weaker specs

Suppose you want to change a method – either how its implementation behaves, or the specification itself. There are already clients that depend on the method's current specification. How do you compare the behaviors of two specifications to decide whether it's safe to replace the old spec with the new spec?

A specification S2 is stronger than or equal to a specification S1 if

- S2's precondition is weaker than or equal to S1's,
and
- S2's postcondition is stronger than or equal to S1's, for the states that satisfy S1's precondition.

If this is the case, then an implementation that satisfies S2 can be used to satisfy S1 as well, and it's safe to replace S1 with S2 in your program.

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 them. And you can always strengthen the post-condition, which means making more promises.

For example, this spec for `find` :

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

can be replaced with:

```
static int findOneOrMore,AnyIndex(int[] a, int val)
    requires: val occurs at least once in a
    effects: returns index i such that a[i] = val
```

which has a weaker precondition. This in turn can be replaced with:

```
static int findOneOrMore,FirstIndex(int[] a, int val)
    requires: val occurs at least once in a
    effects: returns lowest index i such that a[i] = val
```

which has a stronger postcondition.

What about this specification:

```
static int findCanBeMissing(int[] a, int val)
    requires: nothing
    effects: returns index i such that a[i] = val,
            or -1 if no such i
```

We'll come back to `find CanBeMissing` in the exercises.

Diagramming specifications

Imagine (very abstractly) the space of all possible Java methods.

`findFirst`

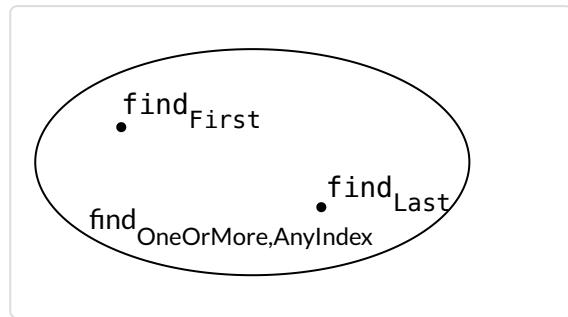
`findLast`

Each point in this space represents a method implementation.

First we'll diagram `find First` and `find Last` defined above. Look back at the code and see that `find First` and `find Last` are *not specs*. They are implementations, with method bodies that implement their actual behavior. So we denote them as points in the space.

A specification defines a *region* in the space of all possible implementations. A given implementation either behaves according to the spec, satisfying the precondition-implies-postcondition contract (it is inside the region), or it does not (outside the region).

Both `find First` and `find Last` satisfy `find OneOrMore,AnyIndex`, so they are inside the region defined by that spec.



We can imagine clients looking in on this space: the specification acts as a firewall.

- Implementors have the freedom to move around inside the spec, changing their code without fear of upsetting a client. This is crucial in order for the implementor to be able to improve the performance of their algorithm, the clarity of their code, or to change their approach when they discover a bug, etc.
- Clients don't know which implementation they will get. They must respect the spec, but also have the freedom to change how they're using the implementation without fear that it will suddenly break.

How will similar specifications relate to one another? Suppose we start with specification S1 and use it to create a new specification S2.

If S2 is stronger than S1, how will these specs appear in our diagram?

- Let's start by **strengthening the postcondition**. If S2's postcondition is now stronger than S1's postcondition, then S2 is the stronger specification.

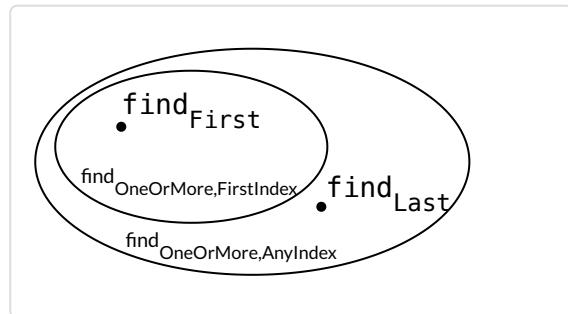
Think about what strengthening the postcondition means for implementors: it means they have less freedom, the requirements on their output are stronger. Perhaps they previously satisfied `find OneOrMore,AnyIndex` by returning any index i , but now the spec demands the *lowest* index i . So there are now implementations *inside* `find OneOrMore,AnyIndex` but *outside* `find OneOrMore,FirstIndex`.

Could there be implementations *inside* `find OneOrMore,FirstIndex` but *outside* `find OneOrMore,AnyIndex`? No. All of those implementations satisfy a stronger postcondition than what `find OneOrMore,AnyIndex` demands.

- Think through what happens if we **weaken the precondition**, which will again make S2 a stronger specification. Implementations will have to handle new inputs that were previously excluded by the spec. If they behaved badly on those inputs before, we wouldn't have noticed, but now their bad behavior is exposed.

We see that when S2 is stronger than S1, it defines a *smaller* region in this diagram; a weaker specification defines a larger region.

In our figure, since `find Last` iterates from the end of the array `arr`, it does not satisfy `find OneOrMore,FirstIndex` and is outside that region.



Another specification S3 that is neither stronger nor weaker than S1 might overlap (such that there exist implementations that satisfy only S1, only S3, and both S1 and S3) or might be disjoint. In both cases, S1 and S3 are incomparable.

Designing good specifications

What makes a good method? Designing a method means primarily writing a specification.

About the form of the specification: it should obviously be succinct, clear, and well-structured, so that it's easy to read.

The content of the specification, however, is harder to prescribe. There are no infallible rules, but there are some useful guidelines.

The specification should be coherent

The spec shouldn't have lots of different cases. Long argument lists, deeply nested if-statements, and boolean flags are all signs of trouble. Consider this specification:

```
static int sumFind(int[] a, int[] b, int val)
  effects: returns the sum of all indices in arrays a and b at which
            val appears
```

Is this a well-designed procedure? Probably not: it's incoherent, since it does several things (finding in two arrays and summing the indexes) that are not really related. It would be better to use two separate procedures, one that finds the indexes, and the other that sums them.

Here's another example, the `countLongWords` method from *Code Review* ([./04-code-review/#countLongWords](#)) :

```
public static int LONG_WORD_LENGTH = 5;
public static String longestWord;

/**
 * Update longestWord to be the longest element of words, and print
 * the number of elements with length > LONG_WORD_LENGTH to the console.
 * @param words list to search for long words
 */
public static void countLongWords(List<String> words)
```

In addition to terrible use of global variables ([./04-code-review/#dont_use_global_variables](#)) and printing instead of returning ([./04-code-review/#methods_should_return_results_not_print_them](#)), the specification is not coherent — it does two different things, counting words and finding the longest word.

Separating those two responsibilities into two different methods will make them simpler (easy to understand) and more useful in other contexts (ready for change).

The results of a call should be informative

Consider the specification of a method that puts a value in a map, where keys are of some type `K` and values are of some type `V` :

```
static V put (Map<K,V> map, K key, V val)
  requires: val may be null, and map may contain null values
  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 `null`s. 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 for sure that you didn't insert `null`s.

The specification should be strong enough

Of course the spec should give clients a strong enough guarantee in the general case — it needs to satisfy their basic requirements. We must use extra care when specifying the special cases, to make sure they don't undermine what would otherwise be a useful method.

For example, there's no point throwing an 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> list1, List<T> list2)
  effects: adds the elements of list2 to list1,
            unless it encounters a null element,
            at which point it throws a NullPointerException
```

If a `NullPointerException` is thrown, the client is left to figure out on their own which elements of `list2` actually made it to `list1`.

The specification should also 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.

The specification should use *abstract types* where possible

We saw early on in the Java Collections section of *Basic Java* (..02-basic-java/#arraylists_and_linkedslists_creating_lists) that we can distinguish between more abstract notions like a `List` or `Set` and particular implementations like `ArrayList` or `HashSet`.

Writing our specification with *abstract types* gives more freedom to both the client and the implementor. In Java, this often means using an interface type, like `Map` or `Reader`, instead of specific implementation types like `HashMap` or `FileReader`. Consider this specification:

```
static ArrayList<T> reverse(ArrayList<T> list)
  effects: returns a new list which is the reversal of list, i.e.
            newList[i] == list[n-i-1]
            for all 0 <= i < n, where n = list.size()
```

This forces the client to pass in an `ArrayList`, and forces the implementor to return an `ArrayList`, even if there might be alternative `List` implementations that they would rather use. Since the behavior of the specification doesn't depend on anything specific about `ArrayList`, it would be better to write this spec in terms of the more abstract `List`.

Precondition or postcondition?

Another design issue is whether to use a precondition, and if so, whether the method code should attempt to make sure the precondition has been met before proceeding. In fact, the most common use of preconditions is to demand a property precisely because it would be hard or expensive for the method to check it.

As mentioned above, a non-trivial precondition 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. That's why the Java API classes, for example, tend to specify (as a postcondition) that they throw unchecked exceptions when arguments are inappropriate. This approach makes it easier to find the bug or incorrect assumption in the caller code that led to passing bad arguments. In general, it's better to **fail fast**, as close as possible to the site of the bug, rather than let bad values propagate through a program far from their original cause.

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. Instead, like the Java API classes, you should throw an exception.

About access control

Read: [Packages](https://docs.oracle.com/javase/tutorial/java/package/index.html) ([//docs.oracle.com/javase/tutorial/java/package/index.html](https://docs.oracle.com/javase/tutorial/java/package/index.html)) (7 short pages) in the Java Tutorials.

Read: [Controlling Access](https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html) ([//docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html](https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html)) (1 page) in the Java Tutorials.

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

About static vs. instance methods

Read: the `static` keyword ([//www.codeguru.com/java/tij/tij0037.shtml#Heading79](http://www.codeguru.com/java/tij/tij0037.shtml#Heading79)) on CodeGuru.

We have also been using *static* for almost all of our methods, again without much discussion. Static methods are not associated with any particular instance of a class, while *instance* methods (declared without the `static` keyword) must be called on a particular object.

Specifications for instance methods are written just the same way as specifications for static methods, but they will often refer to properties of the instance on which they were called.

For example, by now we're very familiar with this specification:

```
static int find(int[] arr, int val)
  requires: val occurs in arr
  effects:  returns index i such that arr[i] = val
```

Instead of using an `int[]`, what if we had a class `IntArray` designed for storing arrays of integers? The `IntArray` class might provide an instance method with the specification:

```
int find(int val)
  requires: val occurs in this array
  effects:  returns index i such that the value at index i in this array
            is val
```

We'll have much more to say about specifications for instance methods in future classes!

Summary

A specification acts as a crucial firewall between implementor and client — both between people (or the same person at different times) and between code. As we saw last time ([./06-specifications/#summary](#)), it makes separate development possible: the client is free to write code that uses a module without seeing its source code, and the implementor is free to write the implementation code without knowing how it will be used.

Declarative specifications are the most useful in practice. Preconditions (which weaken the specification) make life harder for the client, but applied judiciously they are a vital tool in the software designer's repertoire, allowing the implementor to make necessary assumptions.

As always, our goal is to design specifications that make our software:

- **Safe from bugs** . Without specifications, even the tiniest change to any part of our program could be the tipped domino that knocks the whole thing over. Well-structured, coherent specifications minimize misunderstandings and maximize our ability to write correct code with the help of static checking, careful reasoning, testing, and code review.
- **Easy to understand** . A well-written declarative specification means the client doesn't have to read or understand the code. You've probably never read the code for, say, Python `dict.update` (<https://hg.python.org/cpython/file/7ae156f07a90/Objects/dictobject.c#l1990>) , and doing so isn't nearly as useful to the Python programmer as reading the declarative spec (<https://docs.python.org/3/library/stdtypes.html#dict.update>) .
- **Ready for change** . An appropriately weak specification gives freedom to the implementor, and an appropriately strong specification gives freedom to the client. We can even change the specs themselves, without having to revisit every place they're used, as long as we're only strengthening them: weakening preconditions and strengthening postconditions.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 8: Avoiding Debugging

First Defense: Make Bugs Impossible

Second Defense: Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental Development

Modularity & Encapsulation

Reading 8: Avoiding Debugging

6.005 Prime Objective

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

The topic of today's class is debugging – or rather, how to avoid debugging entirely, or keep it easy when you have to do it.

First Defense: Make Bugs Impossible

The best defense against bugs is to make them impossible by design.

One way that we've already talked about is **static checking** ([./01-static-checking/#static_checking_dynamic_checking_no_checking](#)) . Static checking eliminates many bugs by catching them at compile time.

We also saw some examples of **dynamic checking** in earlier class meetings. For example, Java makes array overflow bugs impossible by catching them dynamically. If you try to use an index outside the bounds of an array or a List, then Java automatically produces an error. Older languages like C and C++ silently allow the bad access, which leads to bugs and security vulnerabilities ([//en.wikipedia.org/wiki/Buffer_overflow](https://en.wikipedia.org/wiki/Buffer_overflow)) .

Immutability (immunity from change) is another design principle that prevents bugs. An *immutable* type is a type whose values can never change once they have been created.

String is an immutable type. There are no methods that you can call on a String that will change the sequence of characters that it represents. Strings can be passed around and shared without fear that they will be modified by other code.

Java also gives us *immutable references* : variables declared with the keyword `final` , which can be assigned once but never reassigned. 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.

Consider this example:

```
final char[] vowels = new char[] { 'a', 'e', 'i', 'o', 'u' };
```

The `vowels` variable is declared final, but is it really unchanging? Which of the following statements will be illegal (caught statically by the compiler), and which will be allowed?

```
vowels = new char[] { 'x', 'y', 'z' };
vowels[0] = 'z';
```

You'll find the answers in the exercise below. Be careful about what `final` means! It only makes the *reference* immutable, not necessarily the *object* that the reference points to.

Second Defense: Localize Bugs

If we can't prevent bugs, we can try to localize them to a small part of the program, so that we don't have to look too hard to find the cause of a bug. When localized to a single method or small module, bugs may be found simply by studying the program text.

We already talked about **fail fast** : the earlier a problem is observed (the closer to its cause), the easier it is to fix.

Let's begin with a simple example:

```
/**
 * @param x requires x >= 0
 * @return approximation to square root of x
 */
public double sqrt(double x) { ... }
```

Now suppose somebody calls `sqrt` with a negative argument. What's the best behavior for `sqrt` ? Since the caller has failed to satisfy the requirement that `x` should be nonnegative, `sqrt` is no longer bound by the terms of its contract, so it is technically free to do whatever it wants: return an arbitrary value, or enter an infinite loop, or melt down the CPU. Since the bad call indicates a bug in the caller, however, the most useful behavior would point out the bug as early as possible. We do this by inserting a runtime assertion that tests the precondition. Here is one way we might write the assertion:

```
/**
 * @param x requires x >= 0
 * @return approximation to square root of x
 */
public double sqrt(double x) {
    if (! (x >= 0)) throw new AssertionError();
    ...
}
```

When the precondition is not satisfied, this code terminates the program by throwing an `AssertionError` exception. The effects of the caller's bug are prevented from propagating.

Checking preconditions is an example of **defensive programming**. Real programs are rarely bug-free. Defensive programming offers a way to mitigate the effects of bugs even if you don't know where they are.

Assertions

It is common practice to define a procedure for these kinds of defensive checks, usually called `assert`:

```
assert (x >= 0);
```

This approach abstracts away from what exactly happens when the assertion fails. The failed assert might exit; it might record an event in a log file; it might email a report to a maintainer.

Assertions have the added benefit of documenting an assumption about the state of the program at that point. To somebody reading your code, `assert (x >= 0)` says "at this point, it should always be true that $x \geq 0$." Unlike a comment, however, an assertion is executable code that enforces the assumption at runtime.

In Java, runtime assertions are a built-in feature of the language. The simplest form of the `assert` statement takes a boolean expression, exactly as shown above, and throws `AssertionError` if the boolean expression evaluates to false:

```
assert x >= 0;
```

An assert statement may also include a description expression, which is usually a string, but may also be a primitive type or a reference to an object. The description is printed in an error message when the assertion fails, so it can be used to provide additional details to the programmer about the cause of the failure. The description follows the asserted expression, separated by a colon. For example:

```
assert (x >= 0) : "x is " + x;
```

If $x == -1$, then this assertion fails with the error message

```
x is -1
```

along with a stack trace that tells you where the assert statement was found in your code and the sequence of calls that brought the program to that point. This information is often enough to get started in finding the bug.

A serious problem with Java assertions is that assertions are off by default .

If you just run your program as usual, none of your assertions will be checked! Java's designers did this because checking assertions can sometimes be costly to performance. For most applications, however, assertions are *not* expensive compared to the rest of the code, and the benefit they provide in bug-checking is worth that small cost in performance.

So you have to enable assertions explicitly by passing `-ea` (which stands for *enable assertions*) to the Java virtual machine. In Eclipse, you enable assertions by going to Run → Run Configurations → Arguments, and putting `-ea` in the VM arguments box. It's best, in fact, to enable them by default by going to Preferences → Java → Installed JREs → Edit → Default VM Arguments, as you hopefully did in the Getting Started (..../getting-started/#config-eclipse) instructions.

It's always a good idea to have assertions turned on when you're running JUnit tests. You can ensure that assertions are enabled using the following test case:

```
@Test(expected=AssertionError.class)
public void testAssertionsEnabled() {
    assert false;
}
```

If assertions are turned on as desired, then `assert false` throws an `AssertionError`. The annotation (`expected=AssertionError.class`) on the test expects and requires this error to be thrown, so the test passes. If assertions are turned off, however, then the body of the test will do nothing, failing to throw the expected exception, and JUnit will mark the test as failing.

Note that the Java `assert` statement is a different mechanism from the JUnit methods `assertTrue()`, `assertEquals()`, etc. They all assert a predicate about your code, but are designed for use in different contexts. The `assert` statement should be used in implementation code, for defensive checks inside the implementation. JUnit `assert...()` methods should be used in JUnit tests, to check the result of a test. The `assert` statements don't run without `-ea`, but the JUnit `assert...()` methods always run.

What to Assert

Here are some things you should assert:

Method argument requirements, like we saw for `sqrt`.

Method return value requirements. This kind of assertion is sometimes called a *self check*. For example, the `sqrt` method might square its result to check whether it is reasonably close to `x`:

```
public double sqrt(double x) {
    assert x >= 0;
    double r;
    ... // compute result r
    assert Math.abs(r*r - x) < .0001;
    return r;
}
```

Covering all cases. If a conditional statement or switch does not cover all the possible cases, it is good practice to use an assertion to block the illegal cases:

```
switch (vowel) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u': return "A";
    default: assert false;
}
```

The assertion in the `default` clause has the effect of asserting that `vowel` must be one of the five vowel letters.

When should you write runtime assertions? As you write the code, not after the fact. When you're writing the code, you have the invariants in mind. If you postpone writing assertions, you're less likely to do it, and you're liable to omit some important invariants.

What Not to Assert

Runtime assertions are not free. They can clutter the code, so they must be used judiciously. Avoid trivial assertions, just as you would avoid uninformative comments. For example:

```
// don't do this:  
x = y + 1;  
assert x == y+1;
```

This assertion doesn't find bugs in your code. It finds bugs in the compiler or Java virtual machine, which are components that you should trust until you have good reason to doubt them. If an assertion is obvious from its local context, leave it out.

Never use assertions to test conditions that are external to your program, such as the existence of files, the availability of the network, or the correctness of input typed by a human user. Assertions test the internal state of your program to ensure that it is within the bounds of its specification. When an assertion fails, it indicates that the program has run off the rails in some sense, into a state in which it was not designed to function properly. Assertion failures therefore indicate bugs. External failures are not bugs, and there is no change you can make to your program in advance that will prevent them from happening. External failures should be handled using exceptions instead.

Many assertion mechanisms are designed so that assertions are executed only during testing and debugging, and turned off when the program is released to users. Java's assert statement behaves this way. The advantage of this approach is that you can write very expensive assertions that would otherwise seriously degrade the performance of your program. For example, a procedure that searches an array using binary search has a requirement that the array be sorted. Asserting this requirement requires scanning through the entire array, however, turning an operation that should run in logarithmic time into one that takes linear time. You should be willing (eager!) to pay this cost during testing, since it makes debugging much easier, but not after the program is released to users.

However, disabling assertions in release has a serious disadvantage. With assertions disabled, a program has far less error checking when it needs it most. Novice programmers are usually much more concerned about the performance impact of assertions than they should be. Most assertions are cheap, so they should not be disabled in the official release.

Since assertions may be disabled, the correctness of your program should never depend on whether or not the assertion expressions are executed. In particular, asserted expressions should not have *side-effects*. For example, if you want to assert that an element removed from a list was actually found in the list, don't write it like this:

```
// don't do this:  
assert list.remove(x);
```

If assertions are disabled, the entire expression is skipped, and `x` is never removed from the list. Write it like this instead:

```
boolean found = list.remove(x);  
assert found;
```

For 6.005, you are required to have assertions turned on, all the time. Make sure you did this in Eclipse, following the instructions in the [Getting Started handout \(..../getting-started/#config-eclipse\)](#). If you don't have assertions turned on, you will be sad, and the staff won't have much sympathy.

Incremental Development

A great way to localize bugs to a tiny part of the program is incremental development. Build only a bit of your program at a time, and test that bit thoroughly before you move on. That way, when you discover a bug, it's more likely to be in the part that you just wrote, rather than anywhere in a huge pile of code.

Our class on testing talked about two techniques that help with this:

- **Unit testing** (../03-testing/#unit_testing_and_stubs) : when you test a module in isolation, you can be confident that any bug you find is in that unit – or maybe in the test cases themselves.
- **Regression testing** (../03-testing/#automated_testing_and_regression_testing) : when you're adding a new feature to a big system, run the regression test suite as often as possible. If a test fails, the bug is probably in the code you just changed.

Modularity & Encapsulation

You can also localize bugs by better software design.

Modularity. Modularity means dividing up a system into components, or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system. The opposite of a modular system is a monolithic system – big and with all of its pieces tangled up and dependent on each other.

A program consisting of a single, very long main() function is monolithic – harder to understand, and harder to isolate bugs in. By contrast, a program broken up into small functions and classes is more modular.

Encapsulation. Encapsulation means building walls around a module (a hard shell or capsule) so that the module is responsible for its own internal behavior, and bugs in other parts of the system can't damage its integrity.

One kind of encapsulation is access control

(//docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html), using public and private to control the visibility and accessibility of your variables and methods. A public variable or method can be accessed by any code (assuming the class containing that variable or method is also public). A private variable or method can only be accessed by code in the same class. Keeping things private as much as possible, especially for variables, provides encapsulation, since it limits the code that could inadvertently cause bugs.

Another kind of encapsulation comes from **variable scope**. The scope of a variable is the portion of the program text over which that variable is defined, in the sense that expressions and statements can refer to the variable. A method parameter's scope is the body of the method. A local variable's scope extends from its declaration to the next closing curly brace. Keeping variable scopes as small as possible makes it much easier to reason about where a bug might be in the program. For example, suppose you have a loop like this:

```
for (i = 0; i < 100; ++i) {
    ...
    doSomeThings();
    ...
}
```

...and you've discovered that this loop keeps running forever – i never reaches 100. Somewhere, somebody is changing i. But where? If i is declared as a global variable like this:

```
public static int i;
...
for (i = 0; i < 100; ++i) {
    ...
    doSomeThings();
    ...
}
```

...then its scope is the entire program. It might be changed anywhere in your program: by `doSomeThings()`, by some other method that `doSomeThings()` calls, by a concurrent thread running some completely different code. But if `i` is instead declared as a local variable with a narrow scope, like this:

```
for (int i = 0; i < 100; ++i) {
    ...
    doSomeThings();
    ...
}
```

...then the only place where `i` can be changed is within the for statement – in fact, only in the ... parts that we've omitted. You don't even have to consider `doSomeThings()`, because `doSomeThings()` doesn't have access to this local variable.

Minimizing the scope of variables is a powerful practice for bug localization. Here are a few rules that are good for Java:

- **Always declare a loop variable in the for-loop initializer.** So rather than declaring it before the loop:

```
int i;
for (i = 0; i < 100; ++i) {
```

which makes the scope of the variable the entire rest of the outer curly-brace block containing this code, you should do this:

```
for (int i = 0; i < 100; ++i) {
```

which makes the scope of `i` limited just to the for loop.

- **Declare a variable only when you first need it, and in the innermost curly-brace block that you can.** Variable scopes in Java are curly-brace blocks, so put your variable declaration in the innermost one that contains all the expressions that need to use the variable. Don't declare all your variables at the start of the function – it makes their scopes unnecessarily large. But note that in languages without static type declarations, like Python and Javascript, the scope of a variable is normally the entire function anyway, so you can't restrict the scope of a variable with curly braces, alas.
- **Avoid global variables.** Very bad idea, especially as programs get large. Global variables are often used as a shortcut to provide a parameter to several parts of your program. It's better to just pass the parameter into the code that needs it, rather than putting it in global space where it can inadvertently reassigned.

Summary

In this reading, we looked at some ways to minimize the cost of debugging:

- Avoid debugging
 - make bugs impossible with techniques like static typing, automatic dynamic checking, and immutable types and references
- Keep bugs confined
 - failing fast with assertions keeps a bug's effects from spreading
 - incremental development and unit testing confine bugs to your recent code
 - scope minimization reduces the amount of the program you have to search

Thinking about our three main measures of code quality:

- **Safe from bugs.** We're trying to prevent them and get rid of them.
- **Easy to understand.** Techniques like static typing, final declarations, and assertions are additional documentation of the assumptions in your code. Variable scope minimization makes it easier for a reader to understand how the variable is used, because there's less code to look at.
- **Ready for change.** Assertions and static typing document the assumptions in an automatically-checkable way, so that when a future programmer changes the code, accidental violations of those assumptions are detected.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 9: Mutability & Immutability

Mutability

Risks of mutation

Aliasing is what makes mutable types risky

Specifications for mutating methods

Iterating over arrays and lists

Mutation undermines an iterator

Mutation and contracts

Useful immutable types

Summary

Reading 9: Mutability & Immutability

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- Understand mutability and mutable objects
- Identify aliasing and understand the dangers of mutability
- Use immutability to improve correctness, clarity, & changeability

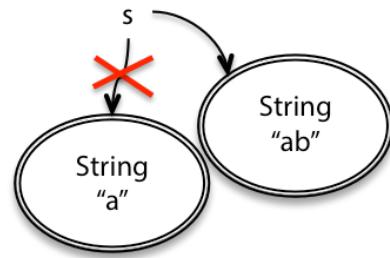
Mutability

Recall from *Basic Java* when we discussed snapshot diagrams ([./02-basic-java/#snapshot_diagrams](#)) that some objects are *immutable* : once created, they always represent the same value. Other objects are *mutable* : they have methods that change the value of the object.

`String` (<https://docs.oracle.com/javase/8/docs/api/?java/lang/String.html>) is an example of an immutable type. A `String` object always represents the same string. `StringBuilder` (<https://docs.oracle.com/javase/8/docs/api/?java/lang/StringBuilder.html>) is an example of a mutable type. It has methods to delete parts of the string, insert or replace characters, etc.

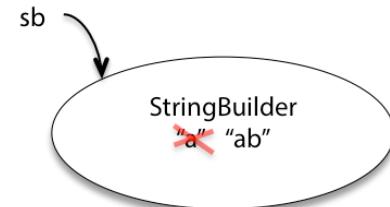
Since `String` is immutable, once created, a `String` object always has the same value. To add something to the end of a `String`, you have to create a new `String` object:

```
String s = "a";
s = s.concat("b"); // s+="b" and s=s+"b" also mean the same thing
```



By contrast, `StringBuilder` objects are mutable. This class has methods that change the value of the object, rather than just returning new values:

```
StringBuilder sb = new StringBuilder("a");
sb.append("b");
```

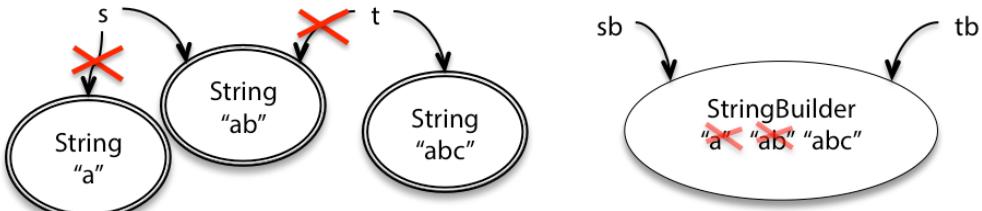


`StringBuilder` has other methods as well, for deleting parts of the string, inserting in the middle, or changing individual characters.

So what? In both cases, you end up with `s` and `sb` referring to the string of characters "ab". 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. For example, when another variable `t` points to the same `String` object as `s`, and another variable `tb` points to the same `StringBuilder` as `sb`, then the differences between the immutable and mutable objects become more evident:

```
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. Consider this code:

```
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` with a `toString()` call:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < n; ++i) {
    sb.append(String.valueOf(i));
}
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.

Risks of mutation

Mutable types seem much more powerful than immutable types. If you were shopping in the Datatype Supermarket, and you had to choose between a boring immutable `String` and a super-powerful-do-anything mutable `StringBuilder`, why on earth would you choose the immutable one?

`StringBuilder` should be able to do everything that `String` can do, plus `set()` and `append()` and everything else.

The answer is that **immutable types are safer from bugs, easier to understand, and more ready for change**. Mutability makes it harder to understand what your program is doing, and much harder to enforce contracts. Here are two examples that illustrate why.

Risky example #1: passing mutable values

Let's start with a simple method that sums the integers in a list:

```
/** @return the sum of the numbers in the list */
public static int sum(List<Integer> list) {
    int sum = 0;
    for (int x : list)
        sum += x;
    return sum;
}
```

Suppose we also need a method that sums the absolute values. Following good DRY practice (Don't Repeat Yourself ([//en.wikipedia.org/wiki/Don't_repeat_yourself](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself))), the implementer writes a method that uses `sum()`:

```
/** @return the sum of the absolute values of the numbers in the list */
public static int sumAbsolute(List<Integer> list) {
    // let's reuse sum(), because DRY, so first we take absolute values
    for (int i = 0; i < list.size(); ++i)
        list.set(i, Math.abs(list.get(i)));
    return sum(list);
}
```

Notice that this method does its job by **mutating the list directly**. It seemed sensible to the implementer, because it's more efficient to reuse the existing list. If the list is millions of items long, then you're saving the time and memory of generating a new million-item list of absolute values. So the implementer has two very good reasons for this design: DRY, and performance.

But the resulting behavior will be very surprising to anybody who uses it! For example:

```
// meanwhile, somewhere else in the code...
public static void main(String[] args) {
    // ...
    List<Integer> myData = Arrays.asList(-5, -3, -2);
    System.out.println(sumAbsolute(myData));
    System.out.println(sum(myData));
}
```

What will this code print? Will it be 10 followed by -10? Or something else?

Let's think about the key points here:

- **Safe from bugs?** In this example, it's easy to blame the implementer of `sumAbsolute()` for going beyond what its spec allowed. But really, **passing mutable objects around is a latent bug**. It's just waiting for some programmer to inadvertently mutate that list, often with very good intentions like reuse or performance, but resulting in a bug that may be very hard to track down.
- **Easy to understand?** When reading `main()`, what would you assume about `sum()` and `sumAbsolute()`? Is it clearly visible to the reader that `myData` gets *changed* by one of them?

Risky example #2: returning mutable values

We just saw an example where passing a mutable object to a function caused problems. What about returning a mutable object?

Let's consider `Date` (<https://docs.oracle.com/javase/8/docs/api/?java/util/Date.html>), one of the built-in Java classes. `Date` happens to be a mutable type. Suppose we write a method that determines the first day of spring:

```
/** @return the first day of spring this year */
public static Date startOfSpring() {
    return askGroundhog();
}
```

Here we're using the well-known Groundhog algorithm for calculating when spring starts (Harold Ramis, Bill Murray, et al. *Groundhog Day*, 1993).

Clients start using this method, for example to plan their big parties:

```
// somewhere else in the code...
public static void partyPlanning() {
    Date partyDate = startOfSpring();
    // ...
}
```

All the code works and people are happy. Now, independently, two things happen. First, the implementer of `startOfSpring()` realizes that the groundhog is starting to get annoyed from being constantly asked when spring will start. So the code is rewritten to ask the groundhog at most once, and then cache the groundhog's answer for future calls:

```
/** @return the first day of spring this year */
public static Date startOfSpring() {
    if (groundhogAnswer == null) groundhogAnswer = askGroundhog();
    return groundhogAnswer;
}
private static Date groundhogAnswer = null;
```

(Aside: note the use of a private static variable for the cached answer. Would you consider this a global variable, or not?)

Second, one of the clients of `startOfSpring()` decides that the actual first day of spring is too cold for the party, so the party will be exactly a month later instead:

```
// somewhere else in the code...
public static void partyPlanning() {
    // let's have a party one month after spring starts!
    Date partyDate = startOfSpring();
    partyDate.setMonth(partyDate.getMonth() + 1);
    // ... uh-oh. what just happened?
}
```

(Aside: this code also has a latent bug in the way it adds a month. Why? What does it implicitly assume about when spring starts?)

What happens when these two decisions interact? Even worse, think about who will first discover this bug — will it be `startOfSpring()`? Will it be `partyPlanning()`? Or will it be some completely innocent third piece of code that also calls `startOfSpring()`?

Key points:

- **Safe from bugs?** Again we had a latent bug that reared its ugly head.
- **Ready for change?** Obviously the mutation of the date object is a change, but that's not the kind of change we're talking about when we say "ready for change." Instead, the question is whether the code of the program can be easily changed without rewriting a lot of it or introducing bugs. Here we had two apparently independent changes, by different programmers, that interacted to produce a bad bug.

In both of these examples — the `List<Integer>` and the `Date` — the problems would have been completely avoided if the list and the date had been immutable types. The bugs would have been impossible by design.

In fact, you should never use `Date`! Use one of the classes from package `java.time`
`(//docs.oracle.com/javase/8/docs/api/index.html?java/time/package-summary.html)`: `LocalDateTime`
`(//docs.oracle.com/javase/8/docs/api/?java/time/LocalDateTime.html)`, `Instant`
`(//docs.oracle.com/javase/8/docs/api/?java/time/Instant.html)`, etc. All guarantee in their specifications that they are *immutable*.

This example also illustrates why using mutable objects can actually be *bad* for performance. The simplest solution to this bug, which avoids changing any of the specifications or method signatures, is for `startOfSpring()` to always return a *copy* of the groundhog's answer:

```
return new Date(groundhogAnswer.getTime());
```

This pattern is **defensive copying**, and we'll see much more of it when we talk about abstract data types. The defensive copy means `partyPlanning()` can freely stomp all over the returned date without affecting `startOfSpring()`'s cached date. But defensive copying forces `startOfSpring()` to do extra work and use extra space for every *client* — even if 99% of the clients never mutate the date it returns. We may end up with lots of copies of the first day of spring throughout memory. If we used an immutable type instead, then different parts of the program could safely share the same values in memory, so less copying and less memory space is required. Immutability can be more efficient than mutability, because immutable types never need to be defensively copied.

Aliasing is what makes mutable types risky

Actually, using mutable objects is just fine if you are using them entirely locally within a method, and with only one reference to the object. What led to the problem in the two examples we just looked at was having multiple references, also called **aliases**, for the same mutable object.

Walking through the examples with a snapshot diagram will make this clear, but here's the outline:

- In the `List` example, the same list is pointed to by both `list` (in `sum` and `sumAbsolute`) and `myData` (in `main`). One programmer (`sumAbsolute`'s) thinks it's ok to modify the list; another programmer (`main`'s) wants the list to stay the same. Because of the aliases, `main`'s programmer loses.
- In the `Date` example, there are two variable names that point to the `Date` object, `groundhogAnswer` and `partyDate`. These aliases are in completely different parts of the code, under the control of different programmers who may have no idea what the other is doing.

Draw snapshot diagrams on paper first, but your real goal should be to develop the snapshot diagram in your head, so you can visualize what's happening in the code.

Specifications for mutating methods

At this point it should be clear that when a method performs mutation, it is crucial to include that mutation in the method's spec, using the structure we discussed in the previous reading (./06-specifications/specs/#specifications_for_mutating_methods).

(Now we've seen that even when a particular method *doesn't* mutate an object, that object's mutability can still be a source of bugs.)

Here's an example of a mutating method:

```
static void sort(List<String> lst)
    requires: nothing
    effects: puts lst in sorted order, i.e. lst[i] <= lst[j]
             for all 0 <= i < j < lst.size()
```

And an example of a method that does not mutate its argument:

```
static List<String> toLowerCase(List<String> lst)
    requires: nothing
    effects: returns a new list t where t[i] = lst[i].toLowerCase()
```

If the `effects` do not explicitly say that an input can be mutated, then in 6.005 we assume mutation of the input is implicitly disallowed. Virtually all programmers would assume the same thing. Surprise mutations lead to terrible bugs.

Iterating over arrays and lists

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 (./02-basic-java/#iteration) to step through a `List` or array. This code:

```
List<String> lst = ...;
for (String str : lst) {
    System.out.println(str);
}
```

is rewritten by the compiler into something like this:

```
List<String> lst = ...;
Iterator iter = lst.iterator();
while (iter.hasNext()) {
    String str = iter.next();
    System.out.println(str);
}
```

An iterator has two methods:

- `next()` returns the next element in the collection
- `hasNext()` tests whether the iterator has reached the end of the collection.

Note that the `next()` method is a **mutator** method, not only returning an element but also advancing the iterator so that the subsequent call to `next()` will return a different element.

You can also look at the Java API definition of `Iterator` ([//docs.oracle.com/javase/8/docs/api/java/util/Iterator.html](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html)) .

Before we go any further:

You should already have read: **Classes and Objects** ([//docs.oracle.com/javase/tutorial/java/javaOO/index.html](https://docs.oracle.com/javase/tutorial/java/javaOO/index.html)) in the Java Tutorials.

Read: **the `final` keyword** ([//www.codeguru.com/java/tij/tij0071.shtml](https://www.codeguru.com/java/tij/tij0071.shtml)) on CodeGuru.

MyIterator

To better understand how an iterator works, here's a simple implementation of an iterator for `ArrayList<String>` :

`MyIterator` makes use of a few Java language features that are different from the classes we've been writing up to this point. Make sure you've read the linked Java Tutorial sections so that you understand them:

Instance variables ([//docs.oracle.com/javase/tutorial/java/javaOO/variables.html](https://docs.oracle.com/javase/tutorial/java/javaOO/variables.html)) , 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** ([//docs.oracle.com/javase/tutorial/java/javaOO/constructors.html](https://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html)) , which makes a new object instance and initializes its instance variables. Where is the constructor of `MyIterator` ?

The `static` keyword is missing from `MyIterator` 's methods, which means they are **instance methods** ([//docs.oracle.com/javase/tutorial/java/javaOO/methods.html](https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html)) that must be called on an instance of the object, e.g. `iter.next()` .

The **this keyword** ([//docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html](https://docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html)) is used at one point to refer to the **instance object** , in particular to refer to an instance variable (`this.list`). This was done to disambiguate two different variables named `list` (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., `index` actually means `this.index` .

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 (access control ([//docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html](https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html))).

final is used to indicate which of the object's internal variables can be reassigned and which can't. `index` is allowed to change (`next()` updates it as it steps through the list), but `list` 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).

```
/***
 * 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> list;
    private int index;
    // list[index] is the next element that will be returned
    // by next()
    // index == list.size() means no more elements to return

    /**
     * Make an iterator.
     * @param list list to iterate over
     */
    public MyIterator(ArrayList<String> list) {
        this.list = list;
        this.index = 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 index < list.size();
    }

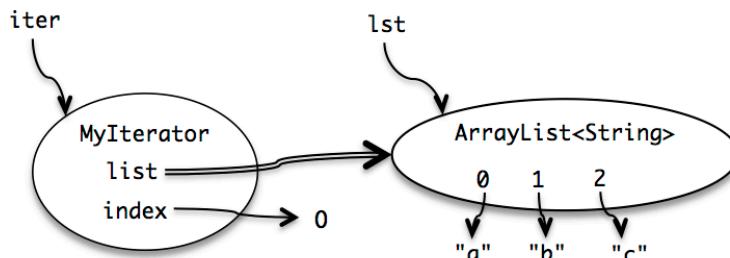
    /**
     * Get the next element of the list.
     * Requires: hasNext() returns true.
     * Modifies: this iterator to advance it to the element
     *           following the returned element.
     * @return next element of the list
     */
    public String next() {
        final String element = list.get(index);
        ++index;
        return element;
    }
}
```

Here's a snapshot diagram showing a typical state for a `MyIterator` object in action:

Note that we draw the arrow from `list` with a double line, to indicate that it's *final*. That means 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 `list` 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 that iterates over it. 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.



Mutation undermines an iterator

Let's try using our iterator for a simple job. Suppose we have a list of MIT subjects represented as strings, like `["6.005", "8.03", "9.00"]`. We want a method `dropCourse6` that will delete the Course 6 subjects from the list, leaving the other subjects behind. Following good practices, we first write the spec:

```

/**
 * Drop all subjects that are from Course 6.
 * Modifies subjects list by removing subjects that start with "6."
 *
 * @param subjects list of MIT subject numbers
 */
public static void dropCourse6(ArrayList<String> subjects)
  
```

Note that `dropCourse6` has a frame condition (the *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:
//   subjects.size: 0, 1, n
//   contents: no 6.xx, one 6.xx, all 6.xx
//   position: 6.xx at start, 6.xx in middle, 6.xx at end

// Test cases:
//   [] => []
//   ["8.03"] => ["8.03"]
//   ["14.03", "9.00", "21L.005"] => ["14.03", "9.00", "21L.005"]
//   ["2.001", "6.01", "18.03"] => ["2.001", "18.03"]
//   ["6.045", "6.005", "6.813"] => []
  
```

Finally, we implement it:

```
public static void dropCourse6(ArrayList<String> subjects) {
    MyIterator iter = new MyIterator(subjects);
    while (iter.hasNext()) {
        String subject = iter.next();
        if (subject.startsWith("6.")) {
            subjects.remove(subject);
        }
    }
}
```

Now we run our test cases, and they work! ... almost. The last test case fails:

```
// dropCourse6(["6.045", "6.005", "6.813"])
// expected [], actual ["6.005"]
```

We got the wrong answer: `dropCourse6` left a course 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 subject : subjects) {
    if (subject.startsWith("6.")) {
        subjects.remove(subject);
    }
}
```

then you'll get a `ConcurrentModificationException` ([//docs.oracle.com/javase/8/docs/api/java/util/ConcurrentModificationException.html](https://docs.oracle.com/javase/8/docs/api/java/util/ConcurrentModificationException.html)). The built-in 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 = subjects.iterator();
while (iter.hasNext()) {
    String subject = iter.next();
    if (subject.startsWith("6.")) {
        iter.remove();
    }
}
```

This is actually more efficient as well, it turns out, because `iter.remove()` already knows where the element it should remove is, while `subjects.remove()` had to search for it again.

But this doesn't fix the whole problem. What if there are other `Iterator`'s currently active over the same list? They won't all be informed!

Mutation and contracts

Mutable objects can make simple contracts very complex

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 the crucial requirement on the client that we've just discovered — that you can't modify a collection while you're iterating over it. Who takes responsibility for it? Iterator

```
//docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html) ? List  
//docs.oracle.com/javase/8/docs/api/?java/util/List.html) ? Collection  
//docs.oracle.com/javase/8/docs/api/?java/util/Collection.html) ? Can you find it?
```

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 a big cost in bug safety for doing so.

Mutable objects reduce changeability

Mutable objects make the contracts between clients and implementers more complicated, and reduce the freedom of the client and implementer to change. In other words, using *objects* that are allowed to change makes the *code* harder to change. Here's an example to illustrate the point.

The crux of our example will be the specification for this method, which looks up a username in MIT's database and returns the user's 9-digit identifier:

```
/**  
 * @param username username of person to look up  
 * @return the 9-digit MIT identifier for username.  
 * @throws NoSuchUserException if nobody with username is in MIT's database  
 */  
public static char[] getMitId(String username) throws NoSuchUserException {  
    // ... look up username in MIT's database and return the 9-digit ID  
}
```

A reasonable specification. Now suppose we have a client using this method to print out a user's identifier:

```
char[] id = getMitId("bitdiddle");  
System.out.println(id);
```

Now both the client and the implementor separately decide to make a change. The client is worried about the user's privacy, and decides to obscure the first 5 digits of the id:

```
char[] id = getMidId("bitdiddle");
for (int i = 0; i < 5; ++i) {
    id[i] = '*';
}
System.out.println(id);
```

The implementer is worried about the speed and load on the database, so the implementer introduces a cache that remembers usernames that have been looked up:

```
private static Map<String, char[]> cache = new HashMap<String, char[]>();

public static char[] getMidId(String username) throws NoSuchUserException {
    // see if it's in the cache already
    if (cache.containsKey(username)) {
        return cache.get(username);
    }

    // ... look up username in MIT's database ...

    // store it in the cache for future lookups
    cache.put(username, id);
    return id;
}
```

These two changes have created a subtle bug. When the client looks up "bitdiddle" and gets back a char array, now both the client and the implementer's cache are pointing to the *same* char array. The array is aliased. That means that the client's obscuring code is actually overwriting the identifier in the cache, so future calls to `getMidId("bitdiddle")` will not return the full 9-digit number, like "928432033", but instead the obscured version "*****2033".

Sharing a mutable object complicates a contract. If this contract failure went to software engineering court, it would be contentious. Who's to blame here? Was the client obliged not to modify the object it got back? Was the implementer obliged not to hold on to the object that it returned?

Here's one way we could have clarified the spec:

```
public static char[] getMidId(String username) throws NoSuchUserException
    requires: nothing
    effects: returns an array containing the 9-digit MIT identifier of username,
            or throws NoSuchUserException if nobody with username is in MIT's
            database. Caller may never modify the returned array.
```

This is a bad way to do it. The problem with this approach is that it means the contract has to be in force for the entire rest of the program. It's a lifetime contract! The other contracts we wrote were much narrower in scope; you could think about the precondition just before the call was made, and the postcondition just after, and you didn't have to reason about what would happen for the rest of time.

Here's a spec with a similar problem:

```
public static char[] getMidId(String username) throws NoSuchUserException
    requires: nothing
    effects: returns a new array containing the 9-digit MIT identifier of username,
            or throws NoSuchUserException if nobody with username is in MIT's
            database.
```

This doesn't entirely fix the problem either. This spec at least says that the array has to be fresh. But does it keep the implementer from holding an alias to that new array? Does it keep the implementer from changing that array or reusing it in the future for something else?

Here's a much better spec:

```
public static String getMitId(String username) throws NoSuchUserException
    requires: nothing
    effects: returns the 9-digit MIT identifier of username, or throws
             NoSuchUserException if nobody with username is in MIT's database.
```

The immutable String return value provides a *guarantee* that the client and the implementer will never step on each other the way they could with char arrays. It doesn't depend on a programmer reading the spec comment carefully. String is *immutable*. Not only that, but this approach (unlike the previous one) gives the implementer the freedom to introduce a cache — a performance improvement.

Useful immutable types

Since immutable types avoid so many pitfalls, let's enumerate some commonly-used immutable types in the Java API:

- The primitive types and primitive wrappers are all immutable. If you need to compute with large numbers, `BigInteger` ([//docs.oracle.com/javase/8/docs/api/?java/math/BigInteger.html](https://docs.oracle.com/javase/8/docs/api/?java/math/BigInteger.html)) and `BigDecimal` ([//docs.oracle.com/javase/8/docs/api/?java/math/BigDecimal.html](https://docs.oracle.com/javase/8/docs/api/?java/math/BigDecimal.html)) are immutable.
- Don't use mutable `Date`s, use the appropriate immutable type from `java.time` ([//docs.oracle.com/javase/8/docs/api/index.html?java/time/package-summary.html](https://docs.oracle.com/javase/8/docs/api/index.html?java/time/package-summary.html)) based on the granularity of timekeeping you need.
- The usual implementations of Java's collections types — `List`, `Set`, `Map` — are all mutable: `ArrayList`, `HashMap`, etc. The `Collections` ([//docs.oracle.com/javase/8/docs/api/?java/util/Collections.html](https://docs.oracle.com/javase/8/docs/api/?java/util/Collections.html)) utility class has methods for obtaining *unmodifiable views* of these mutable collections:
 - `Collections.unmodifiableList`
 - `Collections.unmodifiableSet`
 - `Collections.unmodifiableMap`

You can think of the unmodifiable view as a wrapper around the underlying list/set/map. A client who has a reference to the wrapper and tries to perform mutations — `add`, `remove`, `put`, etc. — will trigger an `UnsupportedOperationException` ([//docs.oracle.com/javase/8/docs/api/?java/lang/UnsupportedOperationException.html](https://docs.oracle.com/javase/8/docs/api/?java/lang/UnsupportedOperationException.html)).

Before we pass a mutable collection to another part of our program, we can wrap it in an unmodifiable wrapper. We should be careful at that point to forget our reference to the mutable collection, lest we accidentally mutate it. (One way to do that is to let it go out of scope.) Just as a mutable object behind a `final` reference can be mutated, the mutable collection inside an unmodifiable wrapper can still be modified by someone with a reference to it, defeating the wrapper.

- `Collections` also provides methods for obtaining immutable empty collections: `Collections.emptyList`, etc. Nothing's worse than discovering your *definitely very empty* list is suddenly *definitely not empty*!

Summary

In this reading, 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.

Make sure you understand the difference between an immutable *object* (like a `String`) and an immutable *reference* (like a `final` variable). Snapshot diagrams can help with this understanding. Objects are values, represented by circles in a snapshot diagram, and an immutable one has a double border indicating that it never changes its value. A reference is a pointer to an object, represented by an arrow in the snapshot diagram, and an immutable reference is an arrow with a double line, indicating that the arrow can't be moved to point to a different object.

The key design principle here is **immutability**: using immutable objects and immutable references as much as possible. Let's review how immutability helps with the main goals of this course:

- **Safe from bugs**. Immutable objects aren't susceptible to bugs caused by aliasing. Immutable references always point to the same object.
- **Easy to understand**. Because an immutable object or reference always means the same thing, it's simpler for a reader of the code to reason about — they don't have to trace through all the code to find all the places where the object or reference might be changed, because it can't be changed.
- **Ready for change**. If an object or reference can't be changed at runtime, then code that depends on that object or reference won't have to be revised when the program changes.

Reading 10: Recursion**Recursion****Choosing the Right Decomposition for a Problem****Structure of Recursive Implementations****Helper Methods****Choosing the Right Recursive Subproblem****Recursive Problems vs. Recursive Data****Reentrant Code****When to Use Recursion Rather Than Iteration****Common Mistakes in Recursive Implementations****Summary**

Reading 10: Recursion

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

After today's class, you should:

- be able to decompose a recursive problem into recursive steps and base cases
- know when and how to use helper methods in recursion
- understand the advantages and disadvantages of recursion vs. iteration

Recursion

In today's class, we're going to talk about how to implement a method, once you already have a specification. We'll focus on one particular technique, *recursion*. Recursion is not appropriate for every problem, but it's an important tool in your software development toolbox, and one that many people scratch their heads over. We want you to be comfortable and competent with recursion, because you will encounter it over and over. (That's a joke, but it's also true.)

Since you've taken 6.01, recursion is not completely new to you, and you have seen and written recursive functions like factorial and fibonacci before. Today's class will delve more deeply into recursion than you may have gone before. Comfort with recursive implementations will be necessary for upcoming classes.

A recursive function is defined in terms of *base cases* and *recursive steps*.

- In a base case, we compute the result immediately given the inputs to the function call.
- In a recursive step, we compute the result with the help of one or more *recursive calls* to this same function, but with the inputs somehow reduced in size or complexity, closer to a base case.

Consider writing a function to compute factorial. We can define factorial in two different ways:

Product	Recurrence relation
$n! = n \times (n - 1) \times \cdots \times 2 \times 1 = \prod_{k=1}^n k$ <small>(where the empty product equals multiplicative identity 1)</small>	$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n - 1)! \times n & \text{if } n > 0 \end{cases}$

which leads to two different implementations:

Iterative	Recursive
<pre>public static long factorial(int n) { long fact = 1; for (int i = 1; i <= n; i++) { fact = fact * i; } return fact; }</pre>	<pre>public static long factorial(int n) { if (n == 0) { return 1; } else { return n * factorial(n-1); } }</pre>

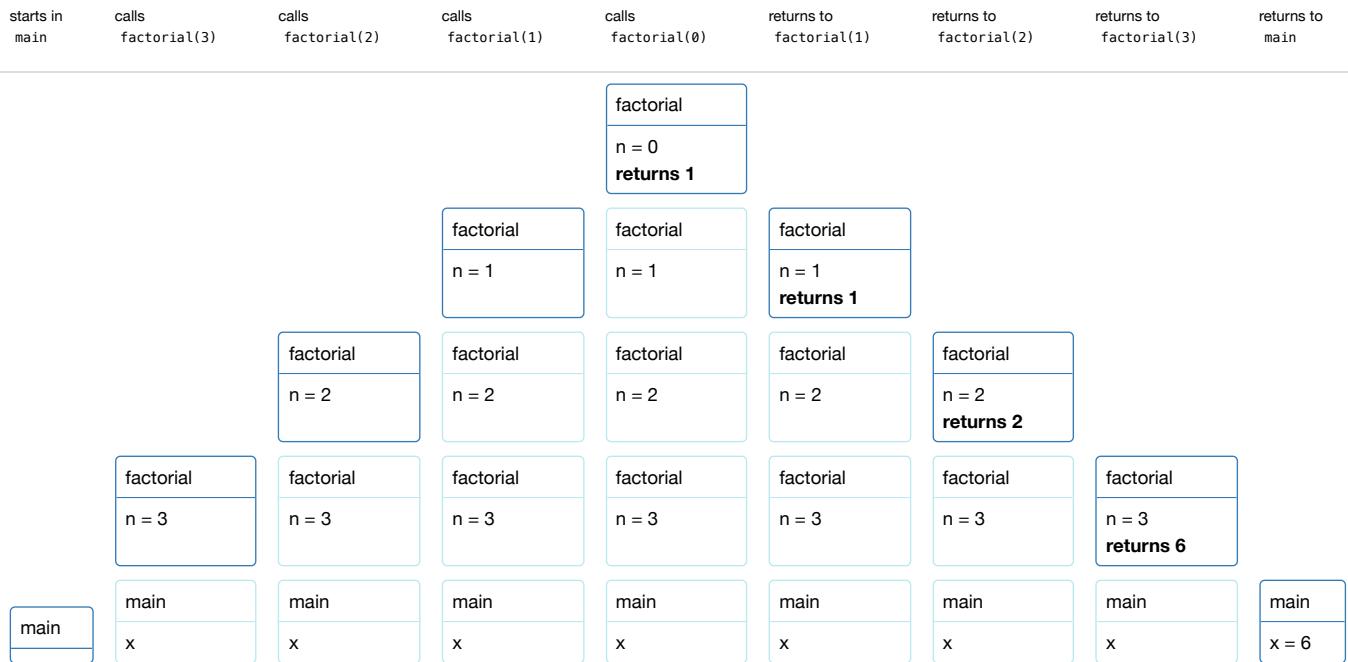
In the recursive implementation on the right, the base case is $n = 0$, where we compute and return the result immediately: $0!$ is defined to be 1. The recursive step is $n > 0$, where we compute the result with the help of a recursive call to obtain $(n-1)!$, then complete the computation by multiplying by n .

To visualize the execution of a recursive function, it is helpful to diagram the *call stack* of currently-executing functions as the computation proceeds.

Let's run the recursive implementation of factorial in a main method:

```
public static void main(String[] args) {
    long x = factorial(3);
}
```

At each step, with time moving left to right:



In the diagram, we can see how the stack grows as main calls factorial and factorial then calls *itself*, until factorial(0) does not make a recursive call. Then the call stack unwinds, each call to factorial returning its answer to the caller, until factorial(3) returns to main .

Here's an [interactive visualization of factorial](#)

([https://www.pythontutor.com/visualize.html#code=public+class+Factorial+%7B%0A++++public+static+void+main\(String%5B%5D+args\)+%7B%0A++++++long+x+%3D%20+n%3D%3D+0\)+%7B%0A++++++return+1%3B%0A++++++%7D+else+%7B%0A++++++return+n-*+factorial\(n-1\)%3B%0A++++++%7D%0A++++%7D%0A%7D&mode=display&origin=opt-frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=0](https://www.pythontutor.com/visualize.html#code=public+class+Factorial+%7B%0A++++public+static+void+main(String%5B%5D+args)+%7B%0A++++++long+x+%3D%20+n%3D%3D+0)+%7B%0A++++++return+1%3B%0A++++++%7D+else+%7B%0A++++++return+n-*+factorial(n-1)%3B%0A++++++%7D%0A++++%7D%0A%7D&mode=display&origin=opt-frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=0)) . You can step through the computation to see the recursion in action. New stack frames grow down instead of up in this visualization.

You've probably seen factorial before, because it's a common example for recursive functions. Another common example is the Fibonacci series:

```
/** 
 * @param n >= 0
 * @return the nth Fibonacci number
 */
public static int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return 1; // base cases
    } else {
        return fibonacci(n-1) + fibonacci(n-2); // recursive step
    }
}
```

Fibonacci is interesting because it has multiple base cases: n=0 and n=1. You can look at an [interactive visualization of Fibonacci](#)

([https://www.pythontutor.com/visualize.html#code=public+class+Fibonacci+%7B%0A++++public+static+void+main\(String%5B%5D+args\)+%7B%0A++++++int+x+%3D-\(n+%3D%3D+0+%7C%7C++%3D%3D+1\)+%7B%0A++++++return+1%3B%0A++++++%7D+else+%7B%0A++++++return+fibonacci\(n-1\)+%2B+fibonacci\(n-2\)%3B%0A++++++%7D%0A++++%7D%0A%7D&mode=display&origin=opt-frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=0](https://www.pythontutor.com/visualize.html#code=public+class+Fibonacci+%7B%0A++++public+static+void+main(String%5B%5D+args)+%7B%0A++++++int+x+%3D-(n+%3D%3D+0+%7C%7C++%3D%3D+1)+%7B%0A++++++return+1%3B%0A++++++%7D+else+%7B%0A++++++return+fibonacci(n-1)+%2B+fibonacci(n-2)%3B%0A++++++%7D%0A++++%7D%0A%7D&mode=display&origin=opt-frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=0)) . Notice that where factorial's stack steadily grows to a maximum depth and then shrinks back to the answer, Fibonacci's stack grows and shrinks repeatedly over the course of the computation.

Choosing the Right Decomposition for a Problem

Finding the right way to decompose a problem, such as a method implementation, is important. Good decompositions are simple, short, easy to understand, safe from bugs, and ready for change.

Recursion is an elegant and simple decomposition for some problems. Suppose we want to implement this specification:

```
/** 
 * @param word consisting only of letters A-Z or a-z
 * @return all subsequences of word, separated by commas,
 * where a subsequence is a string of letters found in word
 * in the same order that they appear in word.
 */
public static String subsequences(String word)
```

For example, `subsequences("abc")` might return `"abc,ab,bc,ac,a,b,c,"`. Note the trailing comma preceding the empty subsequence, which is also a valid subsequence.

This problem lends itself to an elegant recursive decomposition. Take the first letter of the word. We can form one set of subsequences that *include* that letter, and another set of subsequences that exclude that letter, and those two sets completely cover the set of possible subsequences.

```

1 public static String subsequences(String word) {
2     if (word.isEmpty()) {
3         return ""; // base case
4     } else {
5         char firstLetter = word.charAt(0);
6         String restOfWord = word.substring(1);
7
8         String subsequencesOfRest = subsequences(restOfWord);
9
10        String result = "";
11        for (String subsequence : subsequencesOfRest.split(", ", -1)) {
12            result += "," + subsequence;
13            result += "," + firstLetter + subsequence;
14        }
15        result = result.substring(1); // remove extra leading comma
16        return result;
17    }
18}

```

Structure of Recursive Implementations

A recursive implementation always has two parts:

- **base case**, which is the simplest, smallest instance of the problem, that can't be decomposed any further. Base cases often correspond to emptiness – the empty string, the empty list, the empty set, the empty tree, zero, etc.
- **recursive step**, which **decomposes** a larger instance of the problem into one or more simpler or smaller instances that can be solved by recursive calls, and then **recombines** the results of those subproblems to produce the solution to the original problem.

It's important for the recursive step to transform the problem instance into something smaller, otherwise the recursion may never end. If every recursive step shrinks the problem, and the base case lies at the bottom, then the recursion is guaranteed to be finite.

A recursive implementation may have more than one base case, or more than one recursive step. For example, the Fibonacci function has two base cases, $n=0$ and $n=1$.

Helper Methods

The recursive implementation we just saw for `subsequences()` is one possible recursive decomposition of the problem. We took a solution to a subproblem – the subsequences of the remainder of the string after removing the first character – and used it to construct solutions to the original problem, by taking each subsequence and adding the first character or omitting it. This is in a sense a *direct* recursive implementation, where we are using the existing specification of the recursive method to solve the subproblems.

In some cases, it's useful to require a stronger (or different) specification for the recursive steps, to make the recursive decomposition simpler or more elegant. In this case, what if we built up a partial subsequence using the initial letters of the word, and used the recursive calls to *complete* that partial subsequence using the remaining letters of the word? For example, suppose the original word is "orange". We'll both select "o" to be in the partial subsequence, and recursively extend it with all subsequences of "range"; and we'll skip "o", use "" as the partial subsequence, and again recursively extend it with all subsequences of "range".

Using this approach, our code now looks much simpler:

```

/**
 * Return all subsequences of word (as defined above) separated by commas,
 * with partialSubsequence prepended to each one.
 */
private static String subsequencesAfter(String partialSubsequence, String word) {
    if (word.isEmpty()) {
        // base case
        return partialSubsequence;
    } else {
        // recursive step
        return subsequencesAfter(partialSubsequence, word.substring(1))
            + ","
            + subsequencesAfter(partialSubsequence + word.charAt(0), word.substring(1));
    }
}

```

This `subsequencesAfter` method is called a **helper method**. It satisfies a different spec from the original `subsequences`, because it has a new parameter `partialSubsequence`. This parameter fills a similar role that a local variable would in an iterative implementation. It holds temporary state during the evolution of the computation. The recursive calls steadily extend this partial subsequence, selecting or ignoring each letter in the word, until finally reaching the end of the word (the base case), at which point the partial subsequence is returned as the only result. Then the recursion backtracks and fills in other possible subsequences.

To finish the implementation, we need to implement the original `subsequences` spec, which gets the ball rolling by calling the helper method with an initial value for the partial subsequence parameter:

```

public static String subsequences(String word) {
    return subsequencesAfter("", word);
}

```

Don't expose the helper method to your clients. Your decision to decompose the recursion this way instead of another way is entirely implementation-specific. In particular, if you discover that you need temporary variables like `partialSubsequence` in your recursion, don't change the original spec of your method, and don't force your clients to correctly initialize those parameters. That exposes your implementation to the client and reduces your ability to change it in the future. Use a private helper function for the recursion, and have your public method call it with the correct initializations, as shown above.

Choosing the Right Recursive Subproblem

Let's look at another example. Suppose we want to convert an integer to a string representation with a given base, following this spec:

```
/** 
 * @param n integer to convert to string
 * @param base base for the representation. Requires 2<=base<=10.
 * @return n represented as a string of digits in the specified base, with
 *         a minus sign if n<0.
 */
public static String stringValue(int n, int base)
```

For example, `stringValue(16, 10)` should return "16" , and `stringValue(16, 2)` should return "10000" .

Let's develop a recursive implementation of this method. One recursive step here is straightforward: we can handle negative integers simply by recursively calling for the representation of the corresponding positive integer:

```
if (n < 0) return "-" + stringValue(-n, base);
```

This shows that the recursive subproblem can be smaller or simpler in more subtle ways than just the value of a numeric parameter or the size of a string or list parameter. We have still effectively reduced the problem by reducing it to positive integers.

The next question is, given that we have a positive n, say n=829 in base 10, how should we decompose it into a recursive subproblem? Thinking about the number as we would write it down on paper, we could either start with 8 (the leftmost or highest-order digit), or 9 (the rightmost, lower-order digit). Starting at the left end seems natural, because that's the direction we write, but it's harder in this case, because we would need to first find the number of digits in the number to figure out how to extract the leftmost digit. Instead, a better way to decompose n is to take its remainder modulo base (which gives the *rightmost* digit) and also divide by base (which gives the subproblem, the remaining higher-order digits):

```
return stringValue(n/base, base) + "0123456789".charAt(n%base);
```

Think about several ways to break down the problem, and try to write the recursive steps. You want to find the one that produces the simplest, most natural recursive step.

It remains to figure out what the base case is, and include an if statement that distinguishes the base case from this recursive step.

Recursive Problems vs. Recursive Data

The examples we've seen so far have been cases where the problem structure lends itself naturally to a recursive definition. Factorial is easy to define in terms of smaller subproblems. Having a *recursive problem* like this is one cue that you should pull a recursive solution out of your toolbox.

Another cue is when the data you are operating on is inherently recursive in structure. We'll see many examples of recursive data a few classes from now, but for now let's look at the recursive data found in every laptop computer: its filesystem. A filesystem consists of named *files* . Some files are *folders* , which can contain other files. So a filesystem is recursive: folders contain other folders which contain other folders, until finally at the bottom of the recursion are plain (non-folder) files.

The Java library represents the file system using `java.io.File` ([//docs.oracle.com/javase/8/docs/api/index.html?java/io/File.html](https://docs.oracle.com/javase/8/docs/api/index.html?java/io/File.html)) . This is a recursive data type, in the sense that `f.getParentFile()` returns the parent folder of a file `f` , which is a `File` object as well, and `f.listFiles()` returns the files contained by `f` , which is an array of other `File` objects.

For recursive data, it's natural to write recursive implementations:

```
/** 
 * @param f a file in the filesystem
 * @return the full pathname of f from the root of the filesystem
 */
public static String fullPathname(File f) {
    if (f.getParentFile() == null) {
        // base case: f is at the root of the filesystem
        return f.getName();
    } else {
        // recursive step
        return fullPathname(f.getParentFile()) + "/" + f.getName();
    }
}
```

Recent versions of Java have added a new API, `java.nio.Files` ([//docs.oracle.com/javase/8/docs/api/index.html?java/nio/file/Files.html](https://docs.oracle.com/javase/8/docs/api/index.html?java/nio/file/Files.html)) and `java.nio.Path` ([//docs.oracle.com/javase/8/docs/api/index.html?java/nio/file/Path.html](https://docs.oracle.com/javase/8/docs/api/index.html?java/nio/file/Path.html)) , which offer a cleaner separation between the filesystem and the pathnames used to name files in it. But the data structure is still fundamentally recursive.

Reentrant Code

Recursion – a method calling itself – is a special case of a general phenomenon in programming called **reentrancy** . Reentrant code can be safely re-entered, meaning that it can be called again even while a *call to it is underway*. Reentrant code keeps its state entirely in parameters and local variables, and doesn't use static variables or global variables, and doesn't share aliases to mutable objects with other parts of the program, or other calls to itself.

Direct recursion is one way that reentrancy can happen. We've seen many examples of that during this reading. The `factorial()` method is designed so that `factorial(n-1)` can be called even though `factorial(n)` hasn't yet finished working.

Mutual recursion between two or more functions is another way this can happen – A calls B, which calls A again. Direct mutual recursion is virtually always intentional and designed by the programmer. But unexpected mutual recursion can lead to bugs.

When we talk about concurrency later in the course, reentrancy will come up again, since in a concurrent program, a method may be called at the same time by different parts of the program that are running concurrently.

It's good to design your code to be reentrant as much as possible. Reentrant code is safer from bugs and can be used in more situations, like concurrency, callbacks, or mutual recursion.

When to Use Recursion Rather Than Iteration

We've seen two common reasons for using recursion:

- The problem is naturally recursive (e.g. Fibonacci)
- The data is naturally recursive (e.g. filesystem)

Another reason to use recursion is to take more advantage of immutability. In an ideal recursive implementation, all variables are final, all data is immutable, and the recursive methods are all pure functions in the sense that they do not mutate anything. The behavior of a method can be understood simply as a relationship between its parameters and its return value, with no side effects on any other part of the program. This kind of paradigm is called *functional programming*, and it is far easier to reason about than *imperative programming* with loops and variables.

In iterative implementations, by contrast, you inevitably have non-final variables or mutable objects that are modified during the course of the iteration. Reasoning about the program then requires thinking about snapshots of the program state at various points in time, rather than thinking about pure input/output behavior.

One downside of recursion is that it may take more space than an iterative solution. Building up a stack of recursive calls consumes memory temporarily, and the stack is limited in size, which may become a limit on the size of the problem that your recursive implementation can solve.

Common Mistakes in Recursive Implementations

Here are two common ways that a recursive implementation can go wrong:

- The base case is missing entirely, or the problem needs more than one base case but not all the base cases are covered.
- The recursive step doesn't reduce to a smaller subproblem, so the recursion doesn't converge.

Look for these when you're debugging.

On the bright side, what would be an infinite loop in an iterative implementation usually becomes a `StackOverflowError` in a recursive implementation. A buggy recursive program fails faster.

Summary

We saw these ideas:

- recursive problems and recursive data
- comparing alternative decompositions of a recursive problem
- using helper methods to strengthen a recursive step
- recursion vs. iteration

The topics of today's reading connect to our three key properties of good software as follows:

- **Safe from bugs.** Recursive code is simpler and often uses immutable variables and immutable objects.
- **Easy to understand.** Recursive implementations for naturally recursive problems and recursive data are often shorter and easier to understand than iterative solutions.
- **Ready for change.** Recursive code is also naturally reentrant, which makes it safer from bugs and ready to use in more situations.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 11: Debugging

Reproduce the Bug

Understand the Location and Cause of the Bug

Fix the Bug

Reading 11: Debugging

6.005 Prime Objective

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

The topic of today's class is systematic debugging.

Sometimes you have no choice but to debug – particularly when the bug is found only when you plug the whole system together, or reported by a user after the system is deployed, in which case it may be hard to localize it to a particular module. For those situations, we can suggest a systematic strategy for more effective debugging.

Reproduce the Bug

Start by finding a small, repeatable test case that produces the failure. If the bug was found by regression testing, then you're in luck; you already have a failing test case in your test suite. If the bug was reported by a user, it may take some effort to reproduce the bug. For graphical user interfaces and multithreaded programs, a bug may be hard to reproduce consistently if it depends on timing of events or thread execution.

Nevertheless, any effort you put into making the test case small and repeatable will pay off, because you'll have to run it over and over while you search for the bug and develop a fix for it. Furthermore, after you've successfully fixed the bug, you'll want to add the test case to your regression test suite, so that the bug never crops up again. Once you have a test case for the bug, making this test work becomes your goal.

Here's an example. Suppose you have written this function:

```
/**  
 * Find the most common word in a string.  
 * @param text string containing zero or more words, where a word  
 *      is a string of alphanumeric characters bounded by nonalphanumeric.  
 * @return a word that occurs maximally often in text, ignoring alphabetic case.  
 */  
public static String mostCommonWord(String text) {  
    ...  
}
```

A user passes the whole text of Shakespeare's plays into your method, something like `mostCommonWord(allShakespearesPlaysConcatenated)`, and discovers that instead of returning a predictably common English word like "the" or "a", the method returns something unexpected, perhaps "e".

Shakespeare's plays have 100,000 lines containing over 800,000 words, so this input would be very painful to debug by normal methods, like print-debugging and breakpoint-debugging. Debugging will be easier if you first work on reducing the size of the buggy input to something manageable that still exhibits the same (or very similar) bug:

- does the first half of Shakespeare show the same bug? (Binary search! Always a good technique. More about this below.)
- does a single play have the same bug?
- does a single speech have the same bug?

Once you've found a small test case, find and fix the bug using that smaller test case, and then go back to the original buggy input and confirm that you fixed the same bug.

Understand the Location and Cause of the Bug

To localize the bug and its cause, you can use the scientific method:

1. **Study the data.** Look at the test input that causes the bug, and the incorrect results, failed assertions, and stack traces that result from it.
2. **Hypothesize.** Propose a hypothesis, consistent with all the data, about where the bug might be, or where it *cannot* be. It's good to make this hypothesis general at first.
3. **Experiment.** Devise an experiment that tests your hypothesis. It's good to make the experiment an *observation* at first – a probe that collects information but disturbs the system as little as possible.
4. **Repeat.** Add the data you collected from your experiment to what you knew before, and make a fresh hypothesis. Hopefully you have ruled out some possibilities and narrowed the set of possible locations and reasons for the bug.

Let's look at these steps in the context of the `mostCommonWord()` example, fleshed out a little more with three helper methods:

```

/**
 * Find the most common word in a string.
 * @param text string containing zero or more words,
 *             where a word is a string of alphanumeric
 *             characters bounded by nonalphanumeric.
 * @return a word that occurs maximally often in text,
 *         ignoring alphabetic case.
 */
public static String mostCommonWord(String text) {
    ... words = splitIntoWords(text); ...
    ... frequencies = countOccurrences(words); ...
    ... winner = findMostCommon(frequencies); ...
    ... return winner;
}

/** Split a string into words ... */
private static List<String> splitIntoWords(String text) {
    ...
}

/** Count how many times each word appears ... */
private static Map<String, Integer> countOccurrences(List<String> words) {
    ...
}

/** Find the word with the highest frequency count ... */
private static String findMostCommon(Map<String, Integer> frequencies) {
    ...
}

```

1. Study the Data

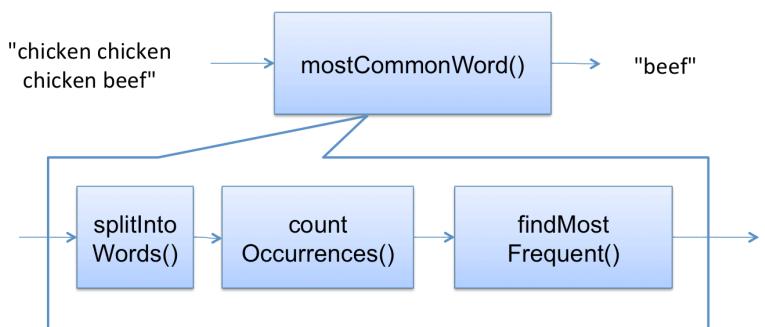
One important form of data is the stack trace from an exception. Practice reading the stack traces that you get, because they will give you enormous amounts of information about where and what the bug might be.

The process of isolating a small test case may also give you data that you didn't have before. You may even have two related test cases that *bracket* the bug in the sense that one succeeds and one fails. For example, maybe `mostCommonWords("c c, b")` is broken, but `mostCommonWords("c c b")` is fine.

2. Hypothesize

It helps to think about your program as modules, or steps in an algorithm, and try to rule out whole sections of the program at once.

The flow of data in `mostCommonWord()` is shown at right. If the symptom of the bug is an exception in `countOccurrences()`, then you can rule out everything downstream, specifically `findMostFrequent()`.



Then you would choose a hypothesis that tries to localize the bug even further. You might hypothesize that the bug is in `splitIntoWords()`, corrupting its results, which then cause the exception in `countOccurrences()`. You would then use an experiment to test that hypothesis. If the hypothesis is true, then you would have ruled out `countOccurrences()` as the source of the problem. If it's false, then you would rule out `splitIntoWords()`.

3. Experiment

A good experiment is a gentle observation of the system without disturbing it much. It might be:

- Run a **different test case**. The test case reduction process discussed above used test cases as experiments.
- Insert a **print statement** or **assertion** in the running program, to check something about its internal state.
- Set a **breakpoint** using a debugger, then single-step through the code and look at variable and object values.

It's tempting to try to insert *fixes* to the hypothesized bug, instead of mere probes. This is almost always the wrong thing to do. First, it leads to a kind of ad-hoc guess-and-test programming, which produces awful, complex, hard-to-understand code. Second, your fixes may just mask the true bug without actually removing it.

For example, if you're getting an `ArrayOutOfBoundsException`, try to understand what's going on first. Don't just add code that avoids or catches the exception, without fixing the real problem.

Other tips

Bug localization by binary search. Debugging is a search process, and you can sometimes use binary search to speed up the process. For example, in `mostCommonWords`, the data flows through three helper methods. To do a binary search, you would divide this workflow in half, perhaps guessing that the bug is found somewhere between the first helper method call and the second, and insert probes (like breakpoints, print statements, or assertions) there to check the results. From the answer to that experiment, you would further divide in half.

Prioritize your hypotheses. When making your hypothesis, you may want to keep in mind that different parts of the system have different likelihoods of failure. For example, old, well-tested code is probably more trustworthy than recently-added code. Java library code is probably more trustworthy than yours. The Java compiler and runtime, operating system platform, and hardware are increasingly more trustworthy, because they are more tried and tested. You should trust these lower levels until you've found good reason not to.

Swap components. If you have another implementation of a module that satisfies the same interface, and you suspect the module, then one experiment you can do is to try swapping in the alternative. For example, if you suspect your `binarySearch()` implementation, then substitute a simpler `linearSearch()` instead. If you suspect `java.util.ArrayList`, you could swap in `java.util.LinkedList` instead. If you suspect the Java runtime, run with a different version of Java. If you suspect the operating system, run your program on a different OS. If you suspect the hardware, run on a different machine. You can waste a lot of time swapping unfailing components, however, so don't do this unless you have good reason to suspect a component.

Make sure your source code and object code are up to date. Pull the latest version from the repository, and delete all your binary files and recompile everything (in Eclipse, this is done by Project → Clean).

Get help. It often helps to explain your problem to someone else, even if the person you're talking to has no idea what you're talking about. Lab assistants and fellow 6.005 students usually do know what you're talking about, so they're even better.

Sleep on it. If you're too tired, you won't be an effective debugger. Trade latency for efficiency.

Fix the Bug

Once you've found the bug and understand its cause, the third step is to devise a fix for it. Avoid the temptation to slap a patch on it and move on. Ask yourself whether the bug was a coding error, like a misspelled variable or interchanged method parameters, or a design error, like an underspecified or insufficient interface. Design errors may suggest that you step back and revisit your design, or at the very least consider all the other clients of the failing interface to see if they suffer from the bug too.

Think also whether the bug has any relatives. If I just found a divide-by-zero error here, did I do that anywhere else in the code? Try to make the code safe from future bugs like this. Also consider what effects your fix will have. Will it break any other code?

Finally, after you have applied your fix, add the bug's test case to your regression test suite, and run all the tests to assure yourself that (a) the bug is fixed, and (b) no new bugs have been introduced.

Summary

In this reading, we looked at how to debug systematically:

- reproduce the bug as a test case, and put it in your regression suite
- find the bug using the scientific method
- fix the bug thoughtfully, not slapdash

Thinking about our three main measures of code quality:

- **Safe from bugs.** We're trying to prevent them and get rid of them.
- **Easy to understand.** Techniques like static typing, final declarations, and assertions are additional documentation of the assumptions in your code. Variable scope minimization makes it easier for a reader to understand how the variable is used, because there's less code to look at.
- **Ready for change.** Assertions and static typing document the assumptions in an automatically-checkable way, so that when a future programmer changes the code, accidental violations of those assumptions are detected.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) | OCW 6.005
 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>)
 Spring 2016

Reading 12: Abstract Data Types

What Abstraction Means

Classifying Types and Operations

Designing an Abstract Type

Representation Independence

Realizing ADT Concepts in Java

Testing an Abstract Data Type

Summary

Reading 12: Abstract Data Types

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

Today's class introduces two ideas:

- Abstract data types
- Representation independence

In this reading, 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 problem: clients making assumptions about the type's internal 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.

Access Control in Java

You should already have read: **Controlling Access to Members of a Class** (<https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>) in the Java Tutorials.

What Abstraction Means

Abstract data types are an instance of a general principle in software engineering, which goes by many names with slightly different shades of meaning. Here are some of the names that are used for this idea:

- **Abstraction.** Omitting or hiding low-level details with a simpler, higher-level idea.
- **Modularity.** Dividing a system into components or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system.
- **Encapsulation.** Building walls around a module (a hard shell or capsule) so that the module is responsible for its own internal behavior, and bugs in other parts of the system can't damage its integrity.

- **Information hiding.** Hiding details of a module's implementation from the rest of the system, so that those details can be changed later without changing the rest of the system.
- **Separation of concerns.** Making a feature (or "concern") the responsibility of a single module, rather than spreading it across multiple modules.

As a software engineer, you should know these terms, because you will run into them frequently. The fundamental purpose of all of these ideas is to help achieve the three important properties that we care about in 6.005: safety from bugs, ease of understanding, and readiness for change.

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, e.g., 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 the original 6.170, the predecessor to 6.005. Barbara Liskov earned the Turing Award, computer science's equivalent of the Nobel Prize, for her work on abstract types.

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.

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 creator 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.
- **Observers** take objects of the abstract type and return objects of a different type. The `size` method of `List`, for example, returns an `int`.
- **Mutators** change objects. The `add` method of `List`, for example, mutates a list by adding an element to the end.

We can summarize these distinctions schematically like this (explanation to follow):

- $\text{creator} : t^* \rightarrow T$

- producer : $T^+ , t^* \rightarrow T$
- observer : $T^+ , t^* \rightarrow t$
- mutator : $T^+ , t^* \rightarrow \text{void} | t | T$

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. The $+$ marker indicates that the type may occur one or more times in that part of the signature, and the $*$ marker indicates that it occurs zero or more times. $|$ indicates or. For example, a producer may take two values of the abstract type T , like `String.concat()`

(<https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#concat-java.lang.String->) does:

- `concat : String × String → String`

Some observers take zero arguments of other types t , such as:

- `size (https://docs.oracle.com/javase/8/docs/api/java/util/List.html#size--) : List → int`

... and others take several:

- `regionMatches (https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#regionMatches-boolean-int-int) : String × boolean × int × String × int × int → boolean`

A creator operation is often implemented as a *constructor*, like `new ArrayList()`

(<https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#ArrayList->). But a creator can simply be a static method instead, like `Arrays.asList()` (<https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList-T-->). A creator implemented as a static method is often called a **factory method**. The various `String.valueOf()` (<https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#valueOf-boolean>) methods in Java are other examples of creators implemented as factory methods.

Mutators are often signaled by a `void` return type. A method that returns `void` *must* be called for some kind of side-effect, since otherwise it doesn't return anything. But not all mutators return `void`. For example, `Set.add()`

(<https://docs.oracle.com/javase/8/docs/api/java/util/Set.html#add-E>) returns a `boolean` that indicates whether the set was actually changed. In Java's graphical user interface toolkit, `Component.add()`

(<https://docs.oracle.com/javase/8/docs/api/java/awt/Container.html#add-javax.awt.Container>) returns the object itself, so that multiple `add()` calls can be chained together (https://en.wikipedia.org/wiki/Method_chaining).

Abstract Data Type Examples

Here are some examples of abstract data types, along with some of their operations, grouped by kind.

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 type. `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 `LinkedList`.

- creators: `ArrayList` and `LinkedList` constructors, `Collections.singletonList` (<https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#singletonList-T>)
- producers: `Collections.unmodifiableList` (<https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#unmodifiableList-T>)
- observers: `size` , `get`
- mutators: `add` , `remove` , `addAll` , `Collections.sort` (<https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#sort-java.util.List->)

String is Java's string type. `String` is immutable.

- creators: `String` 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 reserve the term *producer* only for operations that do no mutation.

Designing an Abstract Type

Designing an abstract type involves choosing good operations and determining how they should behave. Here are a few rules of thumb.

It's better to have **a few, simple operations** that can be combined in powerful ways, rather 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** in the sense that 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. For example, 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

Critically, 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 and postconditions, so that clients know what to depend on, and you know what you can safely change.

Example: Different Representations for Strings

Let's look at a simple abstract data type to see what representation independence means and why it's useful. The `MyString` type below has far fewer operations than the real Java `String`, and their specs are a little different, but it's still illustrative. Here are the specs for the ADT:

```
/** MyString represents an immutable sequence of characters. */
public class MyString {

    ///////////////////// Example of a creator operation ///////////////////
    /** @param b a boolean value
     *  @return string representation of b, either "true" or "false" */
    public static MyString valueOf(boolean b) { ... }

    ///////////////////// Examples of observer operations ///////////////////
    /** @return number of characters in this string */
    public int length() { ... }

    /** @param i character position (requires 0 <= i < string length)
     *  @return character at position i */
    public char charAt(int i) { ... }

    ///////////////////// Example of a producer operation ///////////////////
    /** Get the substring between start (inclusive) and end (exclusive).
     *  @param start starting index
     *  @param end ending index. Requires 0 <= start <= end <= string length.
     *  @return string consisting of charAt(start)...charAt(end-1) */
    public MyString substring(int start, int end) { ... }
}
```

These public operations and their specifications are the only information that a client of this data type is allowed to know. Following the test-first programming paradigm, in fact, the first client we should create is a test suite that exercises these operations according to their specs. At the moment, however, writing test cases that use `assertEquals` directly on `MyString` objects wouldn't work, because we don't have an equality operation defined on `MyString`. We'll talk about how to implement equality carefully in a later reading. For now, the only operations we can perform with `MyStrings` are the ones we've defined above: `valueOf`, `length`, `charAt`, and `substring`. Our tests have to limit themselves to those operations. For example, here's one test for the `valueOf` operation:

```
MyString s = MyString.valueOf(true);
assertEquals(4, s.length());
assertEquals('t', s.charAt(0));
assertEquals('r', s.charAt(1));
assertEquals('u', s.charAt(2));
assertEquals('e', s.charAt(3));
```

We'll come back to the question of testing ADTs at the end of this reading.

For now, let's look at a simple representation for `MyString`: just an array of characters, exactly the length of the string, with no extra room at the end. Here's how that internal representation would be declared, as an instance variable within the class:

```
private char[] a;
```

With that choice of representation, the operations would be implemented in a straightforward way:

```

public static MyString valueOf(boolean b) {
    MyString s = new MyString();
    s.a = b ? new char[] { 't', 'r', 'u', 'e' }
             : new char[] { 'f', 'a', 'l', 's', 'e' };
    return s;
}

public int length() {
    return a.length;
}

public char charAt(int i) {
    return a[i];
}

public MyString substring(int start, int end) {
    MyString that = new MyString();
    that.a = new char[end - start];
    System.arraycopy(this.a, start, that.a, 0, end - start);
    return that;
}

```

Question to ponder: Why don't `charAt` and `substring` have to check whether their parameters are within the valid range? What do you think will happen if the client calls these implementations with illegal inputs?

One problem with this implementation is that it's passing up an opportunity for performance improvement. Because this data type is immutable, the `substring` operation doesn't really have to copy characters out into a fresh array. It could just point to the original `MyString` object's character array and keep track of the start and end that the new `substring` object represents. The `String` implementation in some versions of Java do this.

To implement this optimization, we could change the internal representation of this class to:

```

private char[] a;
private int start;
private int end;

```

With this new representation, the operations are now implemented like this:

```

public static MyString valueOf(boolean b) {
    MyString s = new MyString();
    s.a = b ? new char[] { 't', 'r', 'u', 'e' }
             : new char[] { 'f', 'a', 'l', 's', 'e' };
    s.start = 0;
    s.end = s.a.length;
    return s;
}

public int length() {
    return end - start;
}

public char charAt(int i) {
    return a[start + i];
}

public MyString substring(int start, int end) {
    MyString that = new MyString();
    that.a = this.a;
    that.start = this.start + start;
    that.end = this.start + end;
    return that;
}

```

Because `MyString`'s existing clients depend only on the specs of its public methods, not on its private fields, we can make this change without having to inspect and change all that client code. That's the power of representation independence.

Realizing ADT Concepts in Java

Let's summarize some of the general ideas we've discussed in this reading, which are applicable in general to programming in any language, and their specific realization using Java language features. The point is that there are several ways to do it, and it's important to both understand the big idea, like a creator operation, and different ways to achieve that idea in practice.

The only item in this table that hasn't yet been discussed in this reading is the use of a constant object as a creator operation. This pattern is commonly seen in immutable types, where the simplest or emptiest value of the type is simply a public constant, and producers are used to build up more complex values from it.

ADT concept	Ways to do it in Java	Examples
Creator operation	Constructor	<code>ArrayList()</code> https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#ArrayList--
	Static (factory) method	<code>Collections.singletonList()</code> https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#singletonList-T- , <code>Arrays.asList()</code> https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList-T...-
	Constant	<code>BigInteger.ZERO</code> https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html#ZERO
	Instance method	<code>List.get()</code> (https://docs.oracle.com/javase/8/docs/api/java/util/List.html#get-int-)
	Static method	<code>Collections.max()</code> https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#max-java.util.Collection-
Observer operation		

Producer operation	Instance method	<code>String.trim()</code> (//docs.oracle.com/javase/8/docs/api/java/lang/String.html#trim--)
	Static method	<code>Collections.unmodifiableList()</code> (//docs.oracle.com/javase/8/docs/api/java/util/Collections.html#unmodifiableList--java.util.List-)
Mutator operation	Instance method	<code>List.add()</code> (//docs.oracle.com/javase/8/docs/api/java/util/List.html#add-E-)
	Static method	<code>Collections.copy()</code> (//docs.oracle.com/javase/8/docs/api/java/util/Collections.html#copy--java.util.List-javax.util.List-)
Representation	private fields	

Testing an Abstract Data Type

We build a test suite for an abstract data type by creating tests for each of its operations. These tests inevitably interact with each other. The only way to test creators, producers, and mutators is by calling observers on the objects that result, and likewise, the only way to test observers is by creating objects for them to observe.

Here's how we might partition the input spaces of the four operations in our `MyString` type:

```
// testing strategy for each operation of MyString:
//
// valueOf():
//   true, false
// length():
//   string len = 0, 1, n
//   string = produced by valueOf(), produced by substring()
// charAt():
//   string len = 1, n
//   i = 0, middle, len-1
//   string = produced by valueOf(), produced by substring()
// substring():
//   string len = 0, 1, n
//   start = 0, middle, len
//   end = 0, middle, len
//   end-start = 0, n
//   string = produced by valueOf(), produced by substring()
```

Then a compact test suite that covers all these partitions might look like:

```

@Test public void testValueOfTrue() {
    MyString s = MyString.valueOf(true);
    assertEquals(4, s.length());
    assertEquals('t', s.charAt(0));
    assertEquals('r', s.charAt(1));
    assertEquals('u', s.charAt(2));
    assertEquals('e', s.charAt(3));
}

@Test public void testValueOfFalse() {
    MyString s = MyString.valueOf(false);
    assertEquals(5, s.length());
    assertEquals('f', s.charAt(0));
    assertEquals('a', s.charAt(1));
    assertEquals('l', s.charAt(2));
    assertEquals('s', s.charAt(3));
    assertEquals('e', s.charAt(4));
}

@Test public void testEndSubstring() {
    MyString s = MyString.valueOf(true).substring(2, 4);
    assertEquals(2, s.length());
    assertEquals('u', s.charAt(0));
    assertEquals('e', s.charAt(1));
}

@Test public void testMiddleSubstring() {
    MyString s = MyString.valueOf(false).substring(1, 2);
    assertEquals(1, s.length());
    assertEquals('a', s.charAt(0));
}

@Test public void testSubstringIsWholeString() {
    MyString s = MyString.valueOf(false).substring(0, 5);
    assertEquals(5, s.length());
    assertEquals('f', s.charAt(0));
    assertEquals('a', s.charAt(1));
    assertEquals('l', s.charAt(2));
    assertEquals('s', s.charAt(3));
    assertEquals('e', s.charAt(4));
}

@Test public void testSubstringOfEmptySubstring() {
    MyString s = MyString.valueOf(false).substring(1, 1).substring(0, 0);
    assertEquals(0, s.length());
}

```

Notice that each test case typically calls a few operations that *make* or *modify* objects of the type (creators, producers, mutators) and some operations that *inspect* objects of the type (observers). As a result, each test case covers parts of several operations.

Summary

- Abstract data types are characterized by their operations.
- Operations can be classified into creators, producers, observers, and mutators.
- An ADT's specification is its set of operations and their specs.
- A good ADT is simple, coherent, adequate, and representation-independent.
- An ADT is tested by generating tests for each of its operations, but using the creators, producers, mutators, and observers together in the same tests.

These ideas connect to our three key properties of good software as follows:

- **Safe from bugs.** A good ADT offers a well-defined contract for a data type, so that clients know what to expect from the data type, and implementors have well-defined freedom to vary.
- **Easy to understand.** A good ADT hides its implementation behind a set of simple operations, so that programmers using the ADT only need to understand the operations, not the details of the implementation.
- **Ready for change.** Representation independence allows the implementation of an abstract data type to change without requiring changes from its clients.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 13: Abstraction Functions & Rep Invariants

Invariants

Rep Invariant and Abstraction Function

Documenting the AF, RI, and Safety from Rep Exposure

ADT invariants replace preconditions

Summary

Reading 13: Abstraction Functions & Rep Invariants

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

Today's reading introduces several ideas:

- invariants
- representation exposure
- abstraction functions
- representation invariants

In this reading, we study a more formal mathematical idea of what it means for a class to implement an ADT, via the notions of *abstraction functions* and *rep invariants*. These mathematical notions are eminently practical in software design. The abstraction function will give us a way to cleanly define the equality operation on an abstract data type (which we'll discuss in more depth in a future class). The rep invariant will make it easier to catch bugs caused by a corrupted data structure.

Invariants

Resuming our discussion of what makes a good abstract data type, the final, and perhaps most important, property of a good abstract data type is that it **preserves its own invariants**. An *invariant* is a property of a program that is always true, for every possible runtime state of the program. 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. Saying that the ADT *preserves its own invariants* means that the ADT is responsible for ensuring that its own invariants hold. It doesn't depend on good behavior from its clients.

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 that uses Strings. Contrast that with a string type 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

Later in this reading, we'll see many interesting invariants. Let's focus on immutability for now. Here's a specific example:

```
/*
 * This immutable data type represents a tweet from Twitter.
 */
public class Tweet {

    public String author;
    public String text;
    public Date timestamp;

    /**
     * Make a Tweet.
     * @param author    Twitter user who wrote the tweet
     * @param text      text of the tweet
     * @param timestamp date/time when the tweet was sent
     */
    public Tweet(String author, String text, Date timestamp) {
        this.author = author;
        this.text = text;
        this.timestamp = timestamp;
    }
}
```

How do we guarantee that these Tweet objects are immutable – that, once a tweet is created, its author, message, and date 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:

```
Tweet t = new Tweet("justinbieber",
                     "Thanks to all those believers out there inspiring me every day",
                     new Date());
t.author = "rbmllr";
```

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 Tweet 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 Tweet {

    private final String author;
    private final String text;
    private final Date timestamp;

    public Tweet(String author, String text, Date timestamp) {
        this.author = author;
        this.text = text;
        this.timestamp = timestamp;
    }

    /** @return Twitter user who wrote the tweet */
    public String getAuthor() {
        return author;
    }

    /** @return text of the tweet */
    public String getText() {
        return text;
    }

    /** @return date/time when the tweet was sent */
    public Date getTimestamp() {
        return timestamp;
    }
}

```

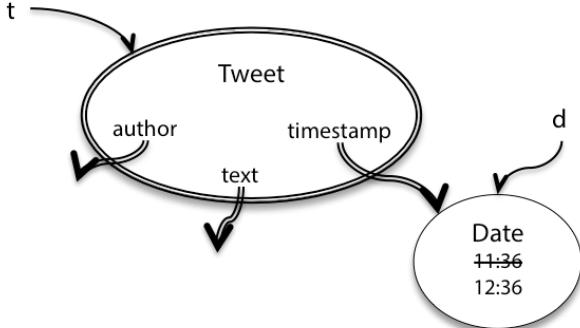
The `private` and `public` keywords indicate which fields and methods are accessible only within the class and which can be accessed from outside the class. The `final` keyword also helps by guaranteeing that the fields of this immutable type won't be reassigned after the object is constructed.

But that's not the end of the story: the rep is still exposed! Consider this perfectly reasonable client code that uses `Tweet` :

```

/** @return a tweet that retweets t, one
 * hour later*/
public static Tweet retweetLater(Tweet t)
{
    Date d = t.getTimestamp();
    d.setHours(d.getHours()+1);
    return new Tweet("rbmllr", t.getText
()), d);
}

```



`retweetLater` takes a tweet and should return another tweet with the same message (called a *retweet*) but sent an hour later. The `retweetLater` method might be part of a system that automatically echoes funny things that Twitter celebrities say.

What's the problem here? The `getTimestamp` call returns a reference to the same `Date` object referenced by tweet `t`. `t.timestamp` and `d` are aliases to the same mutable object. So when that date object is mutated by `d.setHours()`, this affects the date in `t` as well, as shown in the snapshot diagram.

`Tweet`'s immutability invariant has been broken. The problem is that `Tweet` leaked out a reference to a mutable object that its immutability depended on. We exposed the rep, in such a way that `Tweet` can no longer guarantee that its objects are immutable. Perfectly reasonable client code created a subtle bug.

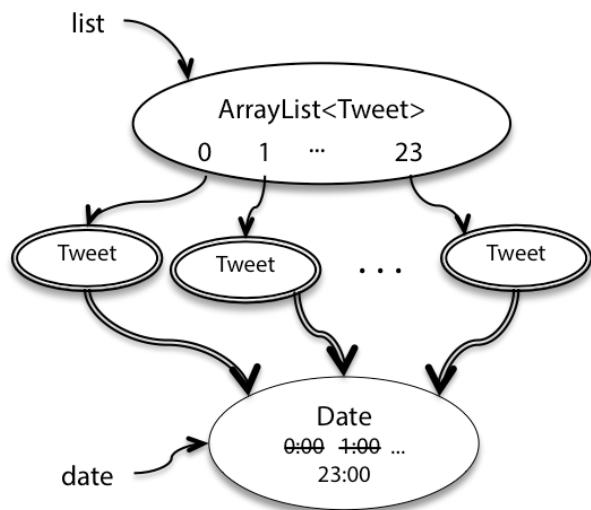
We can patch this kind of rep exposure by using defensive copying: making a copy of a mutable object to avoid leaking out references to the rep. Here's the code:

```
public Date getTimestamp() {
    return new Date(timestamp.getTime());
}
```

Mutable types often have a copy constructor that allows you to make a new instance that duplicates the value of an existing instance. In this case, `Date`'s copy constructor uses the timestamp value, measured in milliseconds since January 1, 1970. As another example, `StringBuilder`'s copy constructor takes a `String`. Another way to copy a mutable object is `clone()`, which is supported by some types but not all. There are unfortunate problems with the way `clone()` works in Java. For more, see Josh Bloch, *Effective Java* ([//library.mit.edu/item/001484188](https://library.mit.edu/item/001484188)), item 11.

So we've done some defensive copying in the return value of `getTimestamp`. But we're not done yet! There's still rep exposure. Consider this (again perfectly reasonable) client code:

```
/** @return a list of 24 inspiring tweets, one per hour today */
public static List<Tweet> tweetEveryHourToday () {
    List<Tweet> list = new ArrayList<Tweet>();
    Date date = new Date();
    for (int i = 0; i < 24; i++) {
        date.setHours(i);
        list.add(new Tweet("rbmllr", "keep it up! you can do it", date));
    }
    return list;
}
```



This code intends to advance a single `Date` object through the 24 hours of a day, creating a tweet for every hour. But notice that the constructor of `Tweet` saves the reference that was passed in, so all 24 `Tweet` objects end up with the same time, as shown in this snapshot diagram.

Again, the immutability of `Tweet` has been violated. We can fix this problem too by using judicious defensive copying, this time in the constructor:

```
public Tweet(String author, String text, Date timestamp) {
    this.author = author;
    this.text = text;
    this.timestamp = new Date(timestamp.getTime());
}
```

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. Doing that creates rep exposure.

You may object that this seems wasteful. Why make all these copies of dates? Why can't we just solve this problem by a carefully written specification, like this?

```
/**  
 * Make a Tweet.  
 * @param author    Twitter user who wrote the tweet  
 * @param text      text of the tweet  
 * @param timestamp date/time when the tweet was sent. Caller must never  
 *                   mutate this Date object again!  
 */  
public Tweet(String author, String text, Date timestamp) {
```

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 – as recommended in *Mutability & Immutability*'s Groundhog Day example ([..09-immutability/#risky_example_2_returning_mutable_values](#)) – we had used an immutable date object, like `java.time.ZonedDateTime`, instead of the mutable `java.util.Date`, then we would have ended this section after talking about `public` and `private`. No further rep exposure would have been possible.

Immutable Wrappers Around Mutable Data Types

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()`, etc. 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, as discussed in *Mutability & Immutability* ([..09-immutability/#useful_immutable_types](#))), 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. But that's still better than nothing, so using unmodifiable lists, maps, and sets can be a very good way to reduce the risk of bugs.

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 representation values (or rep values for short) 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 imaginations. 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.

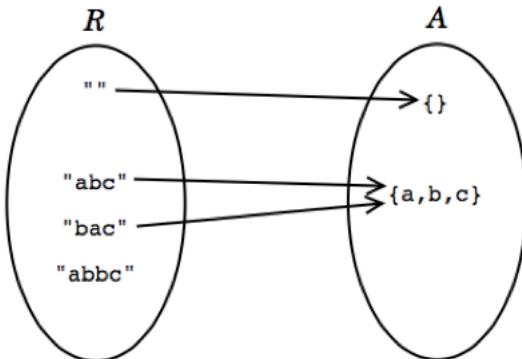
Now of course the implementor of the abstract type must be interested in the representation values, since it is the implementor's job to achieve the illusion of the abstract value space using the rep value space.

Suppose, for example, that we choose to use a string to represent a set of characters:

```
public class CharSet {
    private String s;
    ...
}
```

Then the rep space R contains Strings, and the abstract space A is mathematical sets of characters. 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 picture:

- **Every abstract value is mapped to by some rep value.** 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. In this case, we have decided that the string 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:

1. An *abstraction function* that maps rep values to the abstract values they represent:

```
AF : R → 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 surjective (also called *onto*), not necessarily injective (*one-to-one*) and therefore not necessarily bijective, and often partial.

2. A *rep invariant* that maps rep values to booleans:

```
RI : R → 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.

the abstract space and rep space of CharSet using the NoRepeatsRep

Both the rep invariant and the abstraction function should be documented in the code, right next to the declaration of the rep itself:

```
public class CharSet {
    private String s;
    // Rep invariant:
    //   s contains no repeated characters
    // Abstraction Function:
    //   represents the set of characters found in s
    ...
}
```

A common confusion about abstraction functions and rep invariants is 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.

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 different abstraction 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:

```
public class CharSet {
    private String s;
    // Rep invariant:
    //   s[0] <= s[1] <=
    ... <= s[s.length()-1]
    // Abstraction Function:
    //   represents the set of characters found in s
    ...
}
```

 the abstract space and rep space of CharSet using the SortedRep

Even with the same type for the rep value space and the same rep invariant RI, we might still interpret the rep differently, with different abstraction functions 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 rep "acgg" is interpreted as two range pairs, [a-c] and [g-g], and therefore represents the set {a,b,c,g}. Here's what the AF and RI would look like for that representation:

 the abstract space and rep space of CharSet using the SortedRangeRep

```
public class CharSet {  
    private String s;  
    // Rep invariant:  
    //   s.length is even  
    //   s[0] <= s[1] <= ... <= s[s.length()-1]  
    // Abstraction Function:  
    //   represents the union of the ranges  
    //   {s[i]...s[i+1]} for each adjacent pair of characters  
    //   in s  
    ...  
}
```

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**.

It's critically important to write down these assumptions in your code, as we've done above, so that future programmers (and your future self) are aware of what the representation actually means. Why? What happens if different implementers disagree about the meaning of the rep?

You can find example code for **three different CharSet implementations** (<https://github.com/mit6005/sp16-ex13-adt-examples/tree/master/src/charset>) on GitHub.

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.
     *  @param n value */
    public RatNum(int n) {
        numer = n;
        denom = 1;
        checkRep();
    }

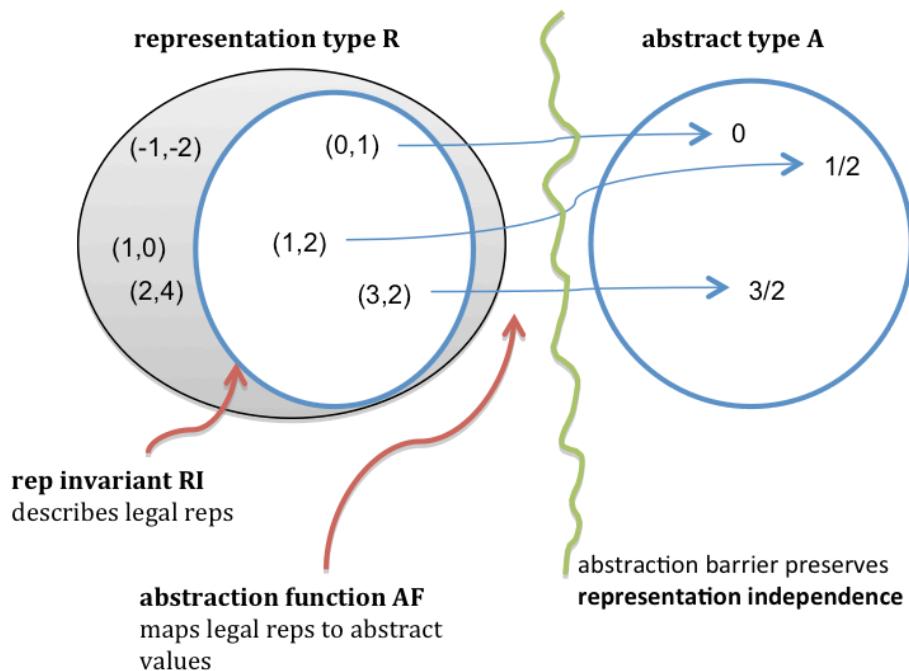
    /** Make a new RatNum == (n / d).
     *  @param n numerator
     *  @param d denominator
     *  @throws ArithmeticException if d == 0 */
    public RatNum(int n, int d) throws ArithmeticException {
        // reduce ratio to lowest terms
        int g = gcd(n, d);
        n = n / g;
        d = d / g;

        // make denominator positive
        if (d < 0) {
            numer = -n;
            denom = -d;
        } else {
            numer = n;
            denom = d;
        }
        checkRep();
    }
}

```

Here is a picture of the abstraction function and rep invariant for this code. The RI requires that numerator/denominator pairs be in reduced form (i.e., lowest terms), so pairs like (2,4) and (18,12) above should be drawn as outside the RI.

It would be completely reasonable to design another implementation of this same ADT with a more permissive RI. With such a change, some operations might become more expensive to perform, and others cheaper.



Checking the Rep Invariant

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. Here's a method for `RatNum` that tests its rep invariant:

```
// 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;
}
```

You should certainly call `checkRep()` to assert the rep invariant at the end of every operation that creates or mutates the rep – in other words, creators, producers, and mutators. Look back at the `RatNum` code above, and you'll see that it calls `checkRep()` at the end of both constructors.

Observer methods don't normally need to call `checkRep()`, but it's good defensive practice to do so anyway. Why? Calling `checkRep()` in every method, including observers, means you'll be more likely to catch rep invariant violations caused by rep exposure.

Why is `checkRep` private? Who should be responsible for checking and enforcing a rep invariant – clients, or the implementation itself?

No Null Values in the Rep

Recall from the *Specifications* reading ([..06-specifications/specs/](#)) that null values are troublesome and unsafe, so much so that we try to remove them from our programming entirely. In 6.005, the preconditions and postconditions of our methods implicitly require that objects and arrays be non-null.

We extend that prohibition to the reps of abstract data types. By default, in 6.005, the rep invariant implicitly includes `x != null` for every reference `x` in the rep that has object type (including references inside arrays or lists). So if your rep is:

```
class CharSet {
    String s;
}
```

then its rep invariant automatically includes `s != null`, and you don't need to state it in a rep invariant comment.

When it's time to implement that rep invariant in a `checkRep()` method, however, you still must *implement* the `s != null` check, and make sure that your `checkRep()` correctly fails when `s` is `null`. Often that check comes for free from Java, because checking other parts of your rep invariant will throw an exception if `s` is null. For example, if your `checkRep()` looks like this:

```
private void checkRep() {
    assert s.length() % 2 == 0;
    ...
}
```

then you don't need `assert s != null`, because the call to `s.length()` will fail just as effectively on a null reference. But if `s` is not otherwise checked by your rep invariant, then `assert s != null` explicitly.

Documenting the AF, RI, and Safety from Rep Exposure

It's good practice to document the abstraction function and rep invariant in the class, using comments right where the private fields of the rep are declared. We've been doing that above.

Another piece of documentation that 6.005 asks you to write is a **rep exposure safety argument**. This is a comment that examines each part of the rep, looks at the code that handles that part of the rep (particularly with respect to parameters and return values from clients, because that is where rep exposure occurs), and presents a reason why the code doesn't expose the rep.

Here's an example of `Tweet` with its rep invariant, abstraction function, and safety from rep exposure fully documented:

```
// Immutable type representing a tweet.
public class Tweet {

    private final String author;
    private final String text;
    private final Date timestamp;

    // Rep invariant:
    //   author is a Twitter username (a nonempty string of letters, digits, underscores)
    //   text.length <= 140
    // Abstraction Function:
    //   represents a tweet posted by author, with content text, at time timestamp

    // Safety from rep exposure:
    //   All fields are private;
    //   author and text are Strings, so are guaranteed immutable;
    //   timestamp is a mutable Date, so Tweet() constructor and getTimestamp()
    //       make defensive copies to avoid sharing the rep's Date object with clients.

    // Operations (specs and method bodies omitted to save space)
    public Tweet(String author, String text, Date timestamp) { ... }
    public String getAuthor() { ... }
    public String getText() { ... }
    public Date getTimestamp() { ... }
}
```

Notice that we don't have any explicit rep invariant conditions on `timestamp` (aside from the conventional assumption that `timestamp!=null`, which we have for all object references). But we still need to include `timestamp` in the rep exposure safety argument, because the immutability property of the whole type depends on all the fields remaining unchanged.

Compare the argument above with an example of a broken argument involving mutable `Date` objects (<https://github.com/mit6005/sp16-ex13-adt-examples/blob/master/src/Timespan.java>) .

Here are the arguments for `RatNum` .

```
// Immutable type representing a rational number.
public class RatNum {
    private final int numer;
    private final int denom;

    // Rep invariant:
    //   denom > 0
    //   numer/denom is in reduced form, i.e. gcd(|numer|,denom) = 1
    // Abstraction Function:
    //   represents the rational number numer / denom
    // Safety from rep exposure:
    //   All fields are private, and all types in the rep are immutable.

    // Operations (specs and method bodies omitted to save space)
    public RatNum(int n) { ... }
    public RatNum(int n, int d) throws ArithmeticException { ... }
    ...
}
```

Notice that an immutable rep is particularly easy to argue for safety from rep exposure.

You can find the full code for `RatNum` (<https://github.com/mit6005/sp16-ex13-adt-examples/blob/master/src/RatNum.java>) on GitHub.

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.

To make an invariant hold, we need to:

- make the invariant true in the initial state of the object; and
- ensure that all changes to the object keep the invariant true.

Translating this in terms of the types of ADT operations, this means:

- creators and producers must establish the invariant for new object instances; and
- mutators and observers must preserve the invariant.

The risk of rep exposure makes the situation more complicated. If the rep is exposed, then the object might be changed anywhere in the program, not just in the ADT's operations, and we can't guarantee that the invariant still holds after those arbitrary changes. So the full rule for proving invariants is:

Structural induction. If an invariant of an abstract data type is

1. established by creators and producers;
2. preserved by mutators, and observers; and
3. no representation exposure occurs,

then the invariant is true of all instances of the abstract data type.

ADT invariants replace preconditions

Now let's bring a lot of pieces together. An enormous advantage of a well-designed abstract data type is that it encapsulates and enforces properties that we would otherwise have to stipulate in a precondition. For example, instead of a spec like this, with an elaborate precondition:

```
/**  
 * @param set1 is a sorted set of characters with no repeats  
 * @param set2 is likewise  
 * @return characters that appear in one set but not the other,  
 * in sorted order with no repeats  
 */  
static String exclusiveOr(String set1, String set2);
```

We can instead use an ADT that captures the desired property:

```
/** @return characters that appear in one set but not the other */  
static SortedSet<Character> exclusiveOr(SortedSet<Character> set1, SortedSet<Character> set2);
```

This is easier to understand, because the name of the ADT conveys all the programmer needs to know. It's also safer from bugs, because Java static checking comes into play, and the required condition (sorted with no repeats) can be enforced in exactly one place, the `SortedSet` ([//docs.oracle.com/javase/8/docs/api/?java/util/SortedSet.html](https://docs.oracle.com/javase/8/docs/api/?java/util/SortedSet.html)) type.

Many of the places where we used preconditions on the problem sets would have benefited from a custom ADT instead.

Summary

- An invariant is a property that is always true of an ADT object instance, for the lifetime of the object.
- A good ADT preserves its own invariants. Invariants must be established by creators and producers, and preserved by observers and mutators.
- The rep invariant specifies legal values of the representation, and should be checked at runtime with `checkRep()`.
- The abstraction function maps a concrete representation to the abstract value it represents.
- Representation exposure threatens both representation independence and invariant preservation.

The topics of today's reading connect to our three properties of good software as follows:

- **Safe from bugs.** A good ADT preserves its own invariants, so that those invariants are less vulnerable to bugs in the ADT's clients, and violations of the invariants can be more easily isolated within the implementation of the ADT itself. Stating the rep invariant explicitly, and checking it at runtime with `checkRep()`, catches misunderstandings and bugs earlier, rather than continuing on with a corrupt data structure.
- **Easy to understand.** Rep invariants and abstraction functions explicate the meaning of a data type's representation, and how it relates to its abstraction.
- **Ready for change.** Abstract data types separate the abstraction from the concrete representation, which makes it possible to change the representation without having to change client code.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) | OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>)

Spring 2016

Reading 14: Interfaces

Interfaces

Subtypes

Example: MyString

Example: Set

Generic Interfaces

Why Interfaces?

Realizing ADT Concepts in Java

Summary

Reading 14: Interfaces

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

The topic of today's class is interfaces: separating the interface of an abstract data type from its implementation, and using Java `interface` types to enforce that separation.

After today's class, you should be able to define ADTs with interfaces, and write classes that implement interfaces.

Interfaces

Java's `interface` is a useful language mechanism for expressing an abstract data type. An interface in Java is a list of method signatures, but no method bodies. A class *implements* an interface if it declares the interface in its `implements` clause, and provides method bodies for all of the interface's methods. So one way to define an abstract data type in Java is as an interface, with its implementation as a class implementing that interface.

One advantage of this approach is that the interface specifies the contract for the client and nothing more. The interface is all a client programmer needs to read to understand the ADT. The client can't create inadvertent dependencies on the ADT's rep, because instance variables can't be put in an interface at all. The implementation is kept well and truly separated, in a different class altogether.

Another advantage is that multiple different representations of the abstract data type can co-exist in the same program, as different classes implementing the interface. When an abstract data type is represented just as a single class, without an interface, it's harder to have multiple representations. In the `MyString` example from *Abstract Data Types* ([./12-abstract-data-types/#example_different_representations_for_strings](#)) , `MyString` was a single class. We explored two different representations for `MyString` , but we couldn't have both representations for the ADT in the same program.

Java's static type checking allows the compiler to catch many mistakes in implementing an ADT's contract. For instance, it is a compile-time error to omit one of the required methods, or to give a method the wrong return type. Unfortunately, the compiler doesn't check for us that the code adheres to the specs of those methods that are

written in documentation comments.

For the details of how to define interfaces in Java, consult the **Java Tutorials section on interfaces** ([//docs.oracle.com/javase/tutorial/java/landl/createinterface.html](https://docs.oracle.com/javase/tutorial/java/landl/createinterface.html)) .

Subtypes

Recall that a *type* is a set of values. The Java `List` ([//docs.oracle.com/javase/8/docs/api/?java/util/List.html](https://docs.oracle.com/javase/8/docs/api/?java/util/List.html)) type is defined by an interface. If we think about all possible `List` values, none of them are `List` objects: we cannot create instances of an interface. Instead, those values are all `ArrayList` objects, or `LinkedList` objects, or objects of another class that implements `List`. A *subtype* is simply a subset of the *supertype*: `ArrayList` and `LinkedList` are subtypes of `List`.

“B is a subtype of A” means “every B is an A.” In terms of specifications: “every B satisfies the specification for A.”

That means B is only a subtype of A if B’s specification is at least as strong as A’s specification. When we declare a class that implements an interface, the Java compiler enforces part of this requirement automatically: for example, it ensures that every method in A appears in B, with a compatible type signature. Class B cannot implement interface A without implementing all of the methods declared in A.

But the compiler cannot check that we haven’t weakened the specification in other ways: strengthening the precondition on some inputs to a method, weakening a postcondition, weakening a guarantee that the interface abstract type advertises to clients. If you declare a subtype in Java — implementing an interface is our current focus — then you must ensure that the subtype’s spec is at least as strong as the supertype’s.

Example: MyString

Let’s revisit `MyString` ([./12-abstract-data-types/#example_different_representations_for_strings](#)). Using an interface instead of a class for the ADT, we can support multiple implementations:

```
/** MyString represents an immutable sequence of characters. */
public interface MyString {

    // We'll skip this creator operation for now
    // /** @param b a boolean value
    //  * @return string representation of b, either "true" or "false" */
    // public static MyString valueOf(boolean b) { ... }

    /** @return number of characters in this string */
    public int length();

    /** @param i character position (requires 0 <= i < string length)
     * @return character at position i */
    public char charAt(int i);

    /** Get the substring between start (inclusive) and end (exclusive).
     * @param start starting index
     * @param end ending index. Requires 0 <= start <= end <= string length.
     * @return string consisting of charAt(start)...charAt(end-1) */
    public MyString substring(int start, int end);
}
```

We’ll skip the static `valueOf` method and come back to it in a minute. Instead, let’s go ahead using a different technique from our toolbox of ADT concepts in Java ([./12-abstract-data-types/#realizing_adt_concepts_in_java](#)): constructors.

Here’s our first implementation:

```

public class SimpleMyString implements MyString {

    private char[] a;

    /* Create an uninitialized SimpleMyString. */
    private SimpleMyString() {}

    /** Create a string representation of b, either "true" or "false".
     * @param b a boolean value */
    public SimpleMyString(boolean b) {
        a = b ? new char[] { 't', 'r', 'u', 'e' }
              : new char[] { 'f', 'a', 'l', 's', 'e' };
    }

    @Override public int length() { return a.length; }

    @Override public char charAt(int i) { return a[i]; }

    @Override public MyString substring(int start, int end) {
        SimpleMyString that = new SimpleMyString();
        that.a = new char[end - start];
        System.arraycopy(this.a, start, that.a, 0, end - start);
        return that;
    }
}

```

And here's the optimized implementation:

```

public class FastMyString implements MyString {

    private char[] a;
    private int start;
    private int end;

    /* Create an uninitialized FastMyString. */
    private FastMyString() {}

    /** Create a string representation of b, either "true" or "false".
     * @param b a boolean value */
    public FastMyString(boolean b) {
        a = b ? new char[] { 't', 'r', 'u', 'e' }
              : new char[] { 'f', 'a', 'l', 's', 'e' };
        start = 0;
        end = a.length;
    }

    @Override public int length() { return end - start; }

    @Override public char charAt(int i) { return a[start + i]; }

    @Override public MyString substring(int start, int end) {
        FastMyString that = new FastMyString();
        that.a = this.a;
        that.start = this.start + start;
        that.end = this.start + end;
        return that;
    }
}

```

- Compare these classes to the implementations of `MyString` in *Abstract Data Types* (./12-abstract-data-types/#example_different_representations_for_strings). Notice how the code that previously appeared in static `valueOf` methods now appears in the constructors, slightly changed to refer to the rep of `this`.
- Also notice the use of `@Override` (<https://docs.oracle.com/javase/tutorial/java/annotations/predefined.html>). This annotation informs the compiler that the method must have the same signature as one of the methods in the interface we're implementing. But since the compiler already checks that we've implemented all of the interface methods, the primary value of `@Override` here is for readers of the code: it tells us to look for the spec of that method in the interface. Repeating the spec wouldn't be DRY, but saying nothing at all makes the code harder to understand.
- And notice the private empty constructors we use to make new instances in `substring(..)` before we fill in their reps with data. We didn't have to write these empty constructors before because Java provides them by default when we don't declare any others. Adding the constructors that take `boolean b` means we have to declare the empty constructors explicitly.

Now that we know good ADTs scrupulously preserve their own invariants (./13-abstraction-functions-rep-invariants/#invariants), these do-nothing constructors are a **bad** pattern: they don't assign any values to the rep, and they certainly don't establish any invariants. We should strongly consider revising the implementation. Since `MyString` is immutable, a starting point would be making all the fields `final`.

How will clients use this ADT? Here's an example:

```
MyString s = new FastMyString(true);
System.out.println("The first character is: " + s.charAt(0));
```

This code looks very similar to the code we write to use the Java collections classes:

```
List<String> s = new ArrayList<String>();
...
```

Unfortunately, this pattern **breaks the abstraction barrier** we've worked so hard to build between the abstract type and its concrete representations. Clients must know the name of the concrete representation class. Because interfaces in Java cannot contain constructors, they must directly call one of the concrete class' constructors. The spec of that constructor won't appear anywhere in the interface, so there's no static guarantee that different implementations will even provide the same constructors.

Fortunately, (as of Java 8) interfaces are allowed to contain static methods, so we can implement the creator operation `valueOf` as a static factory method in the interface `MyString`:

```
public interface MyString {

    /** @param b a boolean value
     *  @return string representation of b, either "true" or "false" */
    public static MyString valueOf(boolean b) {
        return new FastMyString(true);
    }

    // ...
}
```

Now a client can use the ADT without breaking the abstraction barrier:

```
MyString s = MyString.valueOf(true);
System.out.println("The first character is: " + s.charAt(0));
```

Example: Set

Java's collection classes provide a good example of the idea of separating interface and implementation.

Let's consider as an example one of the ADTs from the Java collections library, `Set`. `Set` is the ADT of finite sets of elements of some other type `E`. Here is a simplified version of the `Set` interface:

```
/** A mutable set.
 * @param <E> type of elements in the set */
public interface Set<E> {
```

`Set` is an example of a *generic type*: a type whose specification is in terms of a placeholder type to be filled in later. Instead of writing separate specifications and implementations for `Set<String>`, `Set<Integer>`, and so on, we design and implement one `Set<E>`.

We can match Java interfaces with our classification of ADT operations, starting with a creator:

```
// example creator operation
/** Make an empty set.
 * @param <E> type of elements in the set
 * @return a new set instance, initially empty */
public static <E> Set<E> make() { ... }
```

The `make` operation is implemented as a static factory method. Clients will write code like:

```
Set<String> strings = Set.make();
```

and the compiler will understand that the new `Set` is a set of `String` objects.

```
// example observer operations

/** Get size of the set.
 * @return the number of elements in this set */
public int size();

/** Test for membership.
 * @param e an element
 * @return true iff this set contains e */
public boolean contains(E e);
```

Next we have two observer methods. Notice how the specs are in terms of our abstract notion of a set; it would be malformed to mention the details of any particular implementation of sets with particular private fields. These specs should apply to any valid implementation of the set ADT.

```
// example mutator operations

/** Modifies this set by adding e to the set.
 * @param e element to add */
public void add(E e);

/** Modifies this set by removing e, if found.
 * If e is not found in the set, has no effect.
 * @param e element to remove */
public void remove(E e);
```

The story for these mutators is basically the same as for the observers. We still write specs at the level of our abstract model of sets.

In the Java Tutorials, read these pages:

- **Lesson: Interfaces** ([//docs.oracle.com/javase/tutorial/collections/interfaces/](https://docs.oracle.com/javase/tutorial/collections/interfaces/))
- **The Set Interface** ([//docs.oracle.com/javase/tutorial/collections/interfaces/set.html](https://docs.oracle.com/javase/tutorial/collections/interfaces/set.html))
- **Set Implementations** ([//docs.oracle.com/javase/tutorial/collections/implementations/set.html](https://docs.oracle.com/javase/tutorial/collections/implementations/set.html))
- **The List Interface** ([//docs.oracle.com/javase/tutorial/collections/interfaces/list.html](https://docs.oracle.com/javase/tutorial/collections/interfaces/list.html))

- [List Implementations](https://docs.oracle.com/javase/tutorial/collections/implementations/list.html) ([//docs.oracle.com/javase/tutorial/collections/implementations/list.html](https://docs.oracle.com/javase/tutorial/collections/implementations/list.html))

Generic Interfaces

Suppose we want to implement the generic `Set<E>` interface above.

Generic interface, non-generic implementation. One way we might do this is to implement `Set<E>` for a particular type `E`.

In *Abstraction Functions & Rep Invariants* ([./13-abstraction-functions-rep-invariants/#rep_invariant_and_abstraction_function](#)) we looked at `CharSet`, which represents a set of characters. The example code for `CharSet` (<https://github.com/mit6005/sp16-ex13-adt-examples/tree/master/src/charset>) includes a generic `Set` interface (<https://github.com/mit6005/sp16-ex13-adt-examples/blob/master/src/charset/Set.java>) and each of the implementations `CharSet1 / 2 / 3` declare:

```
public class CharSet implements Set<Character>
```

When the interface mentions placeholder type `E`, the `CharSet` implementations replace `E` with `Character`. For example:

```
public interface Set<E> {
    // ...
    /**
     * Test for membership.
     * @param e an element
     * @return true iff this set contains e
     */
    public boolean contains(E e);

    /**
     * Modifies this set by adding e to the
     * set.
     * @param e element to add
     */
    public void add(E e);
    // ...
}
```

```
public class CharSet1 implements Set<Character> {
    private String s = "";

    // ...

    @Override
    public boolean contains(Character e) {
        checkRep();
        return s.indexOf(e) != -1;
    }

    @Override
    public void add(Character e) {
        if (!contains(e)) s += e;
        checkRep();
    }
    // ...
}
```

The representations used by `CharSet1 / 2 / 3` are not suited for representing sets of arbitrary-type elements. The `String` reps, for example, cannot represent a `Set<Integer>` without careful work to define a new rep invariant and abstraction function that handles multi-digit numbers.

Generic interface, generic implementation. We can also implement the generic `Set<E>` interface without picking a type for `E`. In that case, we write our code blind to the actual type that clients will choose for `E`. Java's `HashSet` ([//docs.oracle.com/javase/8/docs/api/?java/util/HashSet.html](https://docs.oracle.com/javase/8/docs/api/?java/util/HashSet.html)) does that for `Set`. Its declaration looks like:

```
public interface Set<E> {
    // ...
```

```
public class HashSet<E> implements Set<E> {
    // ...
```

A generic implementation can only rely on details of the placeholder types that are included in the interface's specification. We'll see in a future reading how `HashSet` relies on methods that every type in Java is required to implement — and only on those methods, because it can't rely on methods declared in any specific type.

Why Interfaces?

Interfaces are used pervasively in real Java code. Not every class is associated with an interface, but there are a few good reasons to bring an interface into the picture.

- **Documentation for both the compiler and for humans**. Not only does an interface help the compiler catch ADT implementation bugs, but it is also much more useful for a human to read than the code for a concrete implementation. Such an implementation intersperses ADT-level types and specs with implementation details.
- **Allowing performance trade-offs**. Different implementations of the ADT can provide methods with very different performance characteristics. Different applications may work better with different choices, but we would like to code these applications in a way that is representation-independent. From a correctness standpoint, it should be possible to drop in any new implementation of a key ADT with simple, localized code changes.
- **Optional methods**. `List` from the Java standard library marks all mutator methods as optional. By building an implementation that does not support these methods, we can provide immutable lists. Some operations are hard to implement with good enough performance on immutable lists, so we want mutable implementations, too. Code that doesn't call mutators can be written to work automatically with either kind of list.
- **Methods with intentionally underdetermined specifications**. An ADT for finite sets could leave unspecified the element order one gets when converting to a list. Some implementations might use slower method implementations that manage to keep the set representation in some sorted order, allowing quick conversion to a sorted list. Other implementations might make many methods faster by not bothering to support conversion to sorted lists.
- **Multiple views of one class**. A Java class may implement multiple methods. For instance, a user interface widget displaying a drop-down list is natural to view as both a widget and a list. The class for this widget could implement both interfaces. In other words, we don't implement an ADT multiple times just because we are choosing different data structures; we may make multiple implementations because many different sorts of objects may also be seen as special cases of the ADT, among other useful perspectives.
- **More and less trustworthy implementations**. Another reason to implement an interface multiple times might be that it is easy to build a simple implementation that you believe is correct, while you can work harder to build a fancier version that is more likely to contain bugs. You can choose implementations for applications based on how bad it would be to get bitten by a bug.

Realizing ADT Concepts in Java

We can now extend our Java toolbox of ADT concepts ([./12-abstract-data-types/#realizing_adt_concepts_in_java](#)) from the first ADTs reading:

ADT concept	Ways to do it in Java	Examples
Abstract data type	Single class	<code>String</code> (//docs.oracle.com/javase/8/docs/api/java/lang/String.html)
	Interface + class(es)	<code>List</code> (//docs.oracle.com/javase/8/docs/api/java/util/List.html) and <code>ArrayList</code> (//docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)
Creator operation	Constructor	<code>ArrayList()</code> (//docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#ArrayList--)
	Static (factory) method	<code>Collections.singletonList()</code> (//docs.oracle.com/javase/8/docs/api/java/util/Collections.html#singletonList-T-) , <code>Arrays.toList()</code> (//docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList-T...-)
	Constant	<code>BigInteger.ZERO</code> (//docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html#ZERO)
Observer operation	Instance method	<code>List.get()</code> (//docs.oracle.com/javase/8/docs/api/java/util/List.html#get-int-)
	Static method	<code>Collections.max()</code> (//docs.oracle.com/javase/8/docs/api/java/util/Collections.html#max-java.util.Collection-)

Producer operation	Instance method	<code>String.trim()</code> (//docs.oracle.com/javase/8/docs/api/java/lang/String.html#trim--)
	Static method	<code>Collections.unmodifiableList()</code> (//docs.oracle.com/javase/8/docs/api/java/util/Collections.html#unmodifiableList--java.util.List)
Mutator operation	Instance method	<code>List.add()</code> (//docs.oracle.com/javase/8/docs/api/java/util/List.html#add-E)
	Static method	<code>Collections.copy()</code> (//docs.oracle.com/javase/8/docs/api/java/util/Collections.html#copy--java.util.List-java.util.List)
Representation	private fields	

Summary

Java interfaces help us formalize the idea of an abstract data type as a set of operations that must be supported by a type.

This helps make our code...

- **Safe from bugs.** An ADT is defined by its operations, and interfaces do just that. When clients use an interface type, static checking ensures that they only use methods defined by the interface. If the implementation class exposes other methods — or worse, has visible representation — the client can't accidentally see or depend on them. When we have multiple implementations of a data type, interfaces provide static checking of the method signatures.
- **Easy to understand.** Clients and maintainers know exactly where to look for the specification of the ADT. Since the interface doesn't contain instance fields or implementations of instance methods, it's easier to keep details of the implementation out of the specifications.
- **Ready for change.** We can easily add new implementations of a type by adding classes that implement interface. If we avoid constructors in favor of static factory methods, clients will only see the interface. That means we can switch which implementation class clients are using without changing their code at all.

Reading 15: Equality**Introduction****Three Ways to Regard Equality****`==` vs. `equals()`****Equality of Immutable Types****The Object Contract****Equality of Mutable Types****The Final Rule for `equals()` and `hashCode()`****Summary**

Reading 15: Equality

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- Understand equality defined in terms of the abstraction function, an equivalence relation, and observations.
- Differentiate between reference equality and object equality.
- Differentiate between strict observational and behavioral equality for mutable types.
- Understand the Object contract and be able to implement equality correctly for mutable and immutable types.

Introduction

In the previous readings we've developed a rigorous notion of *data abstraction* by creating types that are characterized by their operations, not by their representation. For an abstract data type, the *abstraction function* explains how to interpret a concrete representation value as a value of the abstract type, and we saw how the choice of abstraction function determines how to write the code implementing each of the ADT's operations.

In this reading we turn to how we define the notion of *equality* of values in a data type: the abstraction function will give us a way to cleanly define the equality operation on an ADT.

In the physical world, every object is distinct – at some level, even two snowflakes are different, even if the distinction is just the position they occupy in space. (This isn't strictly true of all subatomic particles, but true enough of large objects like snowflakes and baseballs and people.) So two physical objects are never truly "equal" to each other; they only have degrees of similarity.

In the world of human language, however, and in the world of mathematical concepts, you can have multiple names for the same thing. So it's natural to ask when two expressions represent the same thing: $1+2$, $\sqrt{9}$, and 3 are alternative expressions for the same ideal mathematical value.

Three Ways to Regard Equality

Formally, we can regard equality in several ways.

Using an abstraction function. Recall that an abstraction function $f: R \rightarrow A$ maps concrete instances of a data type to their corresponding abstract values. To use f as a definition for equality, we would say that a equals b if and only if $f(a)=f(b)$.

Using a relation. An *equivalence* is a relation $E \subseteq T \times T$ that is:

- reflexive: $E(t,t) \forall t \in T$
- symmetric: $E(t,u) \Rightarrow E(u,t)$
- transitive: $E(t,u) \wedge E(u,v) \Rightarrow E(t,v)$

To use E as a definition for equality, we would say that a equals b if and only if $E(a,b)$.

These two notions are equivalent. An equivalence relation induces an abstraction function (the relation partitions T , so f maps each element to its partition class). The relation induced by an abstraction function is an equivalence relation (check for yourself that the three properties hold).

A third way we can talk about the equality between abstract values is in terms of what an outsider (a client) can observe about them:

Using observation. We can say that two objects are equal when they cannot be distinguished by observation – every operation we can apply produces the same result for both objects. Consider the set expressions $\{1,2\}$ and $\{2,1\}$. Using the observer operations available for sets, cardinality $|...|$ and membership \in , these expressions are indistinguishable:

- $|\{1,2\}| = 2$ and $|\{2,1\}| = 2$
- $1 \in \{1,2\}$ is true, and $1 \in \{2,1\}$ is true
- $2 \in \{1,2\}$ is true, and $2 \in \{2,1\}$ is true
- $3 \in \{1,2\}$ is false, and $3 \in \{2,1\}$ is false
- ... and so on

In terms of abstract data types, "observation" means calling operations on the objects. So two objects are equal if and only if they cannot be distinguished by calling any operations of the abstract data type.

Example: Duration

Here's a simple example of an immutable ADT.

```
public class Duration {
    private final int mins;
    private final int secs;
    // rep invariant:
    //   mins >= 0, secs >= 0
    // abstraction function:
    //   represents a span of time of mins minutes and secs seconds

    /** Make a duration lasting for m minutes and s seconds. */
    public Duration(int m, int s) {
        mins = m; secs = s;
    }
    /** @return length of this duration in seconds */
    public long getLength() {
        return mins*60 + secs;
    }
}
```

Now which of the following values should be considered equal?

```
Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 3);
Duration d3 = new Duration (0, 62);
Duration d4 = new Duration (1, 2);
```

Think in terms of both the abstraction-function definition of equality, and the observational equality definition.

`==` vs. `equals()`

Like many languages, Java has two different operations for testing equality, with different semantics.

- The `==` operator compares references. More precisely, it tests *referential equality*. Two references are `==` if they point to the same storage in memory. In terms of the snapshot diagrams we've been drawing, two references are `==` if their arrows point to the same object bubble.
- The `equals()` operation compares object contents – in other words, *object equality*, in the sense that we've been talking about in this reading. The `equals()` operation has to be defined appropriately for every abstract data type.

For comparison, here are the equality operators in several languages:

	referential equality	object equality
Java	<code>==</code>	<code>equals()</code>
Objective C	<code>==</code>	<code>isEqual:</code>
C#	<code>==</code>	<code>Equals()</code>
Python	<code>is</code>	<code>==</code>
Javascript	<code>==</code>	n/a

Note that `==` unfortunately flips its meaning between Java and Python. Don't let that confuse you: `==` in Java just tests reference identity, it doesn't compare object contents.

As programmers in any of these languages, we can't change the meaning of the referential equality operator. In Java, `==` always means referential equality. But when we define a new data type, it's our responsibility to decide what object equality means for values of the data type, and implement the `equals()` operation appropriately.

Equality of Immutable Types

The `equals()` method is defined by `Object`, and its default implementation looks like this:

```
public class Object {
    ...
    public boolean equals(Object that) {
        return this == that;
    }
}
```

In other words, the default meaning of `equals()` is the same as referential equality. For immutable data types, this is almost always wrong. So you have to **override** the `equals()` method, replacing it with your own implementation.

Here's our first try for `Duration`:

```
public class Duration {
    ...
    // Problematic definition of equals()
    public boolean equals(Duration that) {
        return this.getLength() == that.getLength();
    }
}
```

There's a subtle problem here. Why doesn't this work? Let's try this code:

```
Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 2);
Object o2 = d2;
d1.equals(d2) → true
d1.equals(o2) → false
```

You can see this code in action

```
(/www.pythontutor.com/java.html#code=public+class+Duration+%7B%0A++++private+final+int+mins%3B%0A++++private+final+int+secs%3B%0A++++//+rep+invar
(1,+2%29%3B%0A+++++Duration+d2+%3D+new+Duration+
(1,+2%29%3B%0A+++++Object+o2+%3D+d2%3B%0A+++++System.out.println(%22d1.equals(d2)%29%3D%22+%2B+d1.equals(d2)%29%29%3B%0A+++++S)
frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=33). You'll see that even though d2
and o2 end up referring to the very same object in memory, you still get different results for them from equals() .
```

What's going on? It turns out that `Duration` has **overloaded** the `equals()` method, because the method signature was not identical to `Object`'s. We actually have two `equals()` methods in `Duration`: an implicit `equals(Object)` inherited from `Object`, and the new `equals(Duration)`.

```
public class Duration extends Object {
    // explicit method that we declared:
    public boolean equals (Duration that) {
        return this.getLength() == that.getLength();
    }
    // implicit method inherited from Object:
    public boolean equals (Object that) {
        return this == that;
    }
}
```

We've seen overloading since the very beginning of the course in static checking ([./01-static-checking/#types](#)). Recall from the Java Tutorials ([//docs.oracle.com/javase/tutorial/java/javaOO/methods.html](#)) that the compiler selects between overloaded operations using the compile-time type of the parameters. For example, when you use the `/` operator, the compiler chooses either integer division or float division based on whether the arguments are ints or floats. The same compile-time selection happens here. If we pass an `Object` reference, as in `d1.equals(o2)`, we end up calling the `equals(Object)` implementation. If we pass a `Duration` reference, as in `d1.equals(d2)`, we end up calling the `equals(Duration)` version. This happens even though `o2` and `d2` both point to the same object at runtime! Equality has become inconsistent.

It's easy to make a mistake in the method signature, and overload a method when you meant to override it. This is such a common error that Java has a language feature, the annotation `@Override` ([https://docs.oracle.com/javase/tutorial/java/annotations/predefined.html](#)), which you should use whenever your intention is to override a method in your superclass. With this annotation, the Java compiler will check that a method with the same signature actually exists in the superclass, and give you a compiler error if you've made a mistake in the signature.

So here's the right way to implement `Duration`'s `equals()` method:

```
@Override
public boolean equals (Object thatObject) {
    if (!(thatObject instanceof Duration)) return false;
    Duration thatDuration = (Duration) thatObject;
    return this.getLength() == thatDuration.getLength();
}
```

This fixes the problem:

```
Duration d1 = new Duration(1, 2);
Duration d2 = new Duration(1, 2);
Object o2 = d2;
d1.equals(d2) → true
d1.equals(o2) → true
```

You can see this code in action

```
(/www.pythontutor.com/java.html#code=public+class+Duration+%7B%0A++++private+final+int+mins%3B%0A++++private+final+int+secs%3B%0A++++//+rep+invar
(Object+thatObject%29+%7B%0A++++++if+!((thatObject+instanceof+Duration)%29%29+return+false%3B%0A++++++Duration+thatDuration%3D+
(Duration%29+thatObject%3B%0A++++++return+this.getLength()%29+%3D%3D+thatDuration.getLength()%29%3B%0A++++%7D%0A++++public+static+void+m=
(1,+2%29%3B%0A+++++Duration+d2+%3D+new+Duration+
(1,+2%29%3B%0A+++++Object+o2+%3D+d2%3B%0A+++++System.out.println(%22d1.equals(d2)%29%3D%22+%2B+d1.equals(d2)%29%29%3B%0A+++++S)
frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=49) in the Online Python Tutor.
```

instanceof

The `instanceof` operator ([https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2.html](#)) tests whether an object is an instance of a particular type. Using `instanceof` is dynamic type checking, not the static type checking we vastly prefer. In general, using `instanceof` in object-oriented programming is a bad smell. In 6.005 — and this is another of our rules that holds true in most good Java programming — **instanceof is disallowed anywhere except for implementing equals**. This prohibition also includes other ways of inspecting objects' runtime types. For example, `getClass` ([//docs.oracle.com/javase/8/docs/api/java/lang/Object.html#getClass--](#)) is also disallowed.

We'll see examples of when you might be tempted to use `instanceof`, and how to write alternatives that are safer from bugs and more ready for change, in a future reading.

The Object Contract

The specification of the `Object` class is so important that it is often referred to as *the Object Contract*. The contract can be found in the method specifications for the `Object` class. Here we will focus on the contract for `equals`. When you override the `equals` method, you must adhere to its general contract. It states that:

- `equals` must define an equivalence relation – that is, a relation that is reflexive, symmetric, and transitive;
- `equals` must be consistent: repeated calls to the method must yield the same result provided no information used in `equals` comparisons on the object is modified;
- for a non-null reference `x`, `x.equals(null)` should return false;
- `hashCode` must produce the same result for two objects that are deemed equal by the `equals` method.

Breaking the Equivalence Relation

Let's start with the equivalence relation. We have to make sure that the definition of equality implemented by `equals()` is actually an equivalence relation as defined earlier: reflexive, symmetric, and transitive. If it isn't, then operations that depend on equality (like sets, searching) will behave erratically and unpredictably. You don't want to program with a data type in which sometimes `a.equals(b)`, but `b` doesn't equal `a`. Subtle and painful bugs will result.

Here's an example of how an innocent attempt to make equality more flexible can go wrong. Suppose we wanted to allow for a tolerance in comparing `Duration` objects, because different computers may have slightly unsynchronized clocks:

```
private static final int CLOCK_SKEW = 5; // seconds

@Override
public boolean equals (Object thatObject) {
    if (!(thatObject instanceof Duration)) return false;
    Duration thatDuration = (Duration) thatObject;
    return Math.abs(this.getLength() - thatDuration.getLength()) <= CLOCK_SKEW;
}
```

Which property of the equivalence relation is violated?

Breaking Hash Tables

To understand the part of the contract relating to the `hashCode` method, you'll need to have some idea of how hash tables work. Two very common collection implementations, `HashSet` and `HashMap`, use a hash table data structure, and depend on the `hashCode` method to be implemented correctly for the objects stored in the set and used as keys in the map.

A hash table is a representation for a mapping: an abstract data type that maps keys to values. Hash tables offer constant time lookup, so they tend to perform better than trees or lists. Keys don't have to be ordered, or have any particular property, except for offering `equals` and `hashCode`.

Here's how a hash table works. It contains an array that is initialized to a size corresponding to the number of elements that we expect to be inserted. When a key and a value are presented for insertion, we compute the hashcode of the key, and convert it into an index in the array's range (e.g., by a modulo division). The value is then inserted at that index.

The rep invariant of a hash table includes the fundamental constraint that keys are in the slots determined by their hash codes.

Hashcodes are designed so that the keys will be spread evenly over the indices. But occasionally a conflict occurs, and two keys are placed at the same index. So rather than holding a single value at an index, a hash table actually holds a list of key/value pairs, usually called a *hash bucket*. A key/value pair is implemented in Java simply as an object with two fields. On insertion, you add a pair to the list in the array slot determined by the hash code. For lookup, you hash the key, find the right slot, and then examine each of the pairs until one is found whose key equals the query key.

Now it should be clear why the `Object` contract requires equal objects to have the same hashcode. If two equal objects had distinct hashcodes, they might be placed in different slots. So if you attempt to lookup a value using a key equal to the one with which it was inserted, the lookup may fail.

`Object`'s default `hashCode()` implementation is consistent with its default `equals()`:

```
public class Object {
    ...
    public boolean equals(Object that) { return this == that; }
    public int hashCode() { return /* the memory address of this */; }
}
```

For references `a` and `b`, if `a == b`, then the address of `a ==` the address of `b`. So the `Object` contract is satisfied.

But immutable objects need a different implementation of `hashCode()`. For `Duration`, since we haven't overridden the default `hashCode()` yet, we're currently breaking the `Object` contract:

```
Duration d1 = new Duration(1, 2);
Duration d2 = new Duration(1, 2);
d1.equals(d2) → true
d1.hashCode() → 2392
d2.hashCode() → 4823
```

`d1` and `d2` are `equal()`, but they have different hash codes. So we need to fix that.

A simple and drastic way to ensure that the contract is met is for `hashCode` to always return some constant value, so every object's hash code is the same. This satisfies the `Object` contract, but it would have a disastrous performance effect, since every key will be stored in the same slot, and every lookup will degenerate to a linear search along a long list.

The standard way to construct a more reasonable hash code that still satisfies the contract is to compute a hash code for each component of the object that is used in the determination of equality (usually by calling the `hashCode` method of each component), and then combining these, throwing in a few arithmetic operations. For `Duration`, this is easy, because the abstract value of the class is already an integer value:

```
@Override
public int hashCode() {
    return (int) getLength();
```

Josh Bloch's fantastic book, *Effective Java*, explains this issue in more detail, and gives some strategies for writing decent hash code functions. The advice is summarized in a good StackOverflow post ([//stackoverflow.com/questions/113511/hash-code-implementation](https://stackoverflow.com/questions/113511/hash-code-implementation)). Recent versions of Java now have a utility method `Objects.hash()` ([//docs.oracle.com/javase/8/docs/api/java/util/Objects.html#hash-Java.lang.Object...](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html#hash-Java.lang.Object...)) that makes it easier to implement a hash code involving multiple fields.

Note, however, that as long as you satisfy the requirement that equal objects have the same hash code value, then the particular hashing technique you use doesn't make a difference to the correctness of your code. It may affect its performance, by creating unnecessary collisions between different objects, but even a poorly-performing hash function is better than one that breaks the contract.

Most crucially, note that if you don't override `hashCode` at all, you'll get the one from `Object`, which is based on the address of the object. If you have overridden `equals`, this will mean that you will have almost certainly violated the contract. So as a general rule:

Always override `hashCode` when you override `equals`.

Many years ago in (a precursor to 6.005 confusingly numbered) 6.170, a student spent hours tracking down a bug in a project that amounted to nothing more than misspelling `hashCode` as `hashcode`. This created a new method that didn't override the `hashCode` method of `Object` at all, and strange things happened. Use `@Override`!

Equality of Mutable Types

We've been focusing on equality of immutable objects so far in this reading. What about mutable objects?

Recall our definition: two objects are equal when they cannot be distinguished by observation. With mutable objects, there are two ways to interpret this:

- when they cannot be distinguished by observation *that doesn't change the state of the objects*, i.e., by calling only observer, producer, and creator methods. This is often strictly called **observational equality**, since it tests whether the two objects "look" the same, in the current state of the program.
- when they cannot be distinguished by *any* observation, even state changes. This interpretation allows calling any methods on the two objects, including mutators. This is often called **behavioral equality**, since it tests whether the two objects will "behave" the same, in this and all future states.

For immutable objects, observational and behavioral equality are identical, because there aren't any mutator methods.

For mutable objects, it's tempting to implement strict observational equality. Java uses observational equality for most of its mutable data types, in fact. If two distinct `List` objects contain the same sequence of elements, then `equals()` reports that they are equal.

But using observational equality leads to subtle bugs, and in fact allows us to easily break the rep invariants of other collection data structures. Suppose we make a `List`, and then drop it into a `Set`:

```
List<String> list = new ArrayList<>();
list.add("a");

Set<List<String>> set = new HashSet<List<String>>();
set.add(list);
```

We can check that the set contains the list we put in it, and it does:

```
set.contains(list) → true
```

But now we mutate the list:

```
list.add("goodbye");
```

And it no longer appears in the set!

```
set.contains(list) → false!
```

It's worse than that, in fact: when we iterate over the members of the set, we still find the list in there, but `contains()` says it's not there!

```
for (List<String> l : set) {
    set.contains(l) → false!
}
```

If the set's own iterator and its own `contains()` method disagree about whether an element is in the set, then the set clearly is broken. You can see this code in action

([//www.pythontutor.com/java.html#code=import+java.util.*%3B%0Apublic+class+WhyObservationalEqualityHurts+%7B%0A++public+static+void+main\(String%5B%5E%29%3CString%3E+l+%3A+set%29+%7B%0A++++System.out.println\(%22set.contains\(%29%3D%22+%2B+set.contains\(%29%29%3B%0A++++%7D%0A++frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=13\)](https://www.pythontutor.com/java.html#code=import+java.util.*%3B%0Apublic+class+WhyObservationalEqualityHurts+%7B%0A++public+static+void+main(String%5B%5E%29%3CString%3E+l+%3A+set%29+%7B%0A++++System.out.println(%22set.contains(%29%3D%22+%2B+set.contains(%29%29%3B%0A++++%7D%0A++frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=13))) on Online Python Tutor.

What's going on? `List<String>` is a mutable object. In the standard Java implementation of collection classes like `List`, mutations affect the result of `equals()` and `hashCode()`. When the list is first put into the `HashSet`, it is stored in the hash bucket corresponding to its `hashCode()` result at that time. When the list is subsequently mutated, its `hashCode()` changes, but `HashSet` doesn't realize it should be moved to a different bucket. So it can never be found again.

When `equals()` and `hashCode()` can be affected by mutation, we can break the rep invariant of a hash table that uses that object as a key.

Here's a telling quote from the specification of `java.util.Set`:

Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set.

The Java library is unfortunately inconsistent about its interpretation of `equals()` for mutable classes. Collections use observational equality, but other mutable classes (like `StringBuilder`) use behavioral equality.

The lesson we should draw from this example is that `equals()` should implement behavioral equality. In general, that means that two references should be `equals()` if and only if they are aliases for the same object. So mutable objects should just inherit `equals()` and `hashCode()` from `Object`. For clients that need a notion of observational equality (whether two mutable objects “look” the same in the current state), it’s better to define a new method, e.g., `similar()`.

The Final Rule for `equals()` and `hashCode()`

For immutable types :

- `equals()` should compare abstract values. This is the same as saying `equals()` should provide behavioral equality.
- `hashCode()` should map the abstract value to an integer.

So immutable types must override both `equals()` and `hashCode()`.

For mutable types :

- `equals()` should compare references, just like `==`. Again, this is the same as saying `equals()` should provide behavioral equality.
- `hashCode()` should map the reference into an integer.

So mutable types should not override `equals()` and `hashCode()` at all, and should simply use the default implementations provided by `Object`. Java doesn’t follow this rule for its collections, unfortunately, leading to the pitfalls that we saw above.

Autoboxing and Equality

One more instructive pitfall in Java. We’ve talked about primitive types and their object type equivalents – for example, `int` and `Integer`. The object type implements `equals()` in the correct way, so that if you create two `Integer` objects with the same value, they’ll be `equals()` to each other:

```
Integer x = new Integer(3);
Integer y = new Integer(3);
x.equals(y) → true
```

But there’s a subtle problem here; `==` is overloaded. For reference types like `Integer`, it implements referential equality:

```
x == y // returns false
```

But for primitive types like `int`, `==` implements behavioral equality:

```
(int)x == (int)y // returns true
```

So you can’t really use `Integer` interchangeably with `int`. The fact that Java automatically converts between `int` and `Integer` (this is called *autoboxing* and *autounboxing*) can lead to subtle bugs! You have to be aware what the compile-time types of your expressions are. Consider this:

```
Map<String, Integer> a = new HashMap(), b = new HashMap();
a.put("c", 130); // put ints into the map
b.put("c", 130);
a.get("c") == b.get("c") → ?? // what do we get out of the map?
```

You can see this code in action

([https://www.pythontutor.com/java.html#code=import+java.util.*%3B%0Apublic+class+AutounboxingProblem+%7B%0A++public+static+void+main\(String%5B%5D+args%0A.get\(%22c%22%29+%3D%3D+b.get\(%22c%22%29%29+%29%3B%0A++%7D%0A%7D&mode=display&origin=opt-frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=6](https://www.pythontutor.com/java.html#code=import+java.util.*%3B%0Apublic+class+AutounboxingProblem+%7B%0A++public+static+void+main(String%5B%5D+args%0A.get(%22c%22%29+%3D%3D+b.get(%22c%22%29%29+%29%3B%0A++%7D%0A%7D&mode=display&origin=opt-frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=6)) on Online Python Tutor.

Summary

- Equality should be an equivalence relation (reflexive, symmetric, transitive).
- Equality and hash code must be consistent with each other, so that data structures that use hash tables (like `HashSet` and `HashMap`) work properly.
- The abstraction function is the basis for equality in immutable data types.
- Reference equality is the basis for equality in mutable data types; this is the only way to ensure consistency over time and avoid breaking rep invariants of hash tables.

Equality is one part of implementing an abstract data type, and we’ve already seen how important ADTs are to achieving our three primary objectives. Let’s look at equality in particular:

- **Safe from bugs**. Correct implementation of equality and hash codes is necessary for use with collection data types like sets and maps. It’s also highly desirable for writing tests. Since every object in Java inherits the `Object` implementations, immutable types must override them.
- **Easy to understand**. Clients and other programmers who read our specs will expect our types to implement an appropriate equality operation, and will be surprised and confused if we do not.
- **Ready for change**. Correctly-implemented equality for *immutable* types separates equality of reference from equality of abstract value, hiding from clients our decisions about whether values are shared. Choosing behavioral rather than observational equality for *mutable* types helps avoid unexpected aliasing bugs.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 16: Recursive Data Types

Introduction

Part 1: Recursive Data Types

Part 2: Writing a Program with ADTs

Summary

Reading 16: Recursive Data Types

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- Understand recursive datatypes
- Read and write datatype definitions
- Understand and implement functions over recursive datatypes
- Understand immutable lists and know the standard operations on immutable lists
- Know and follow a recipe for writing programs with ADTs

Introduction

In this reading we'll look at recursively-defined types, how to specify operations on such types, and how to implement them. Our main example will be *immutable lists*.

Then we'll use another recursive datatype example, *matrix multiplications*, to walk through our process for programming with ADTs.

Part 1: Recursive Data Types (recursive/)

Part 2: Writing a Program with ADTs (matexpr/)

Summary

Let's review how recursive datatypes fit in with the main goals of this course:

- **Safe from bugs**. Recursive datatypes allow us to tackle problems with a recursive or unbounded structure. Implementing appropriate data structures that encapsulate important operations and maintain their own invariants is crucial for correctness.

- **Easy to understand** . Functions over recursive datatypes, specified in the abstract type and implemented in each concrete variant, organize the different behavior of the type.
- **Ready for change** . A recursive ADT, like any ADT, separates abstract values from concrete representations, making it possible to change low-level code and high-level structure of the implementation without changing clients.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 17: Regular Expressions & Grammars

Introduction

Grammars

Regular Expressions

Summary

Reading 17: Regular Expressions & Grammars

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

After today's class, you should:

- Understand the ideas of grammar productions and regular expression operators
- Be able to read a grammar or regular expression and determine whether it matches a sequence of characters
- Be able to write a grammar or regular expression to match a set of character sequences and parse them into a data structure

Introduction

Today's reading introduces several ideas:

- grammars, with productions, nonterminals, terminals, and operators
- regular expressions
- parser generators

Some program modules take input or produce output in the form of a sequence of bytes or a sequence of characters, which is called a *string* when it's simply stored in memory, or a *stream* when it flows into or out of a module. In today's reading, we talk about how to write a specification for such a sequence.

Concretely, a sequence of bytes or characters might be:

- A file on disk, in which case the specification is called the *file format*
- Messages sent over a network, in which case the specification is a *wire protocol*
- A command typed by the user on the console, in which case the specification is a *command line interface*
- A string stored in memory

For these kinds of sequences, we introduce the notion of a *grammar*, which allows us not only to distinguish between legal and illegal sequences, but also to parse a sequence into a data structure that a program can work with. The data structure produced from a grammar will often be a recursive data type like we talked about in the recursive data type reading (./16-recursive-data-types/).

We also talk about a specialized form of a grammar called a *regular expression*. In addition to being used for specification and parsing, regular expressions are a widely-used tool for many string-processing tasks that need to disassemble a string, extract information from it, or transform it.

The next reading will talk about parser generators, a kind of tool that translates a grammar automatically into a parser for that grammar.

Grammars

To describe a sequence of symbols, whether they are bytes, characters, or some other kind of symbol drawn from a fixed set, we use a compact representation called a *grammar*.

A *grammar* defines a set of sentences, where each *sentence* is a sequence of symbols. For example, our grammar for URLs will specify the set of sentences that are legal URLs in the HTTP protocol.

The symbols in a sentence are called *terminals* (or tokens).

They're called terminals because they are the leaves of a tree that represents the structure of the sentence. They don't have any children, and can't be expanded any further. We generally write terminals in quotes, like 'http' or ':' .

A grammar is described by a set of *productions*, where each production defines a *nonterminal*. You can think of a nonterminal like a variable that stands for a set of sentences, and the production as the definition of that variable in terms of other variables (nonterminals), operators, and constants (terminals). Nonterminals are internal nodes of the tree representing a sentence.

A production in a grammar has the form

nonterminal ::= expression of terminals, nonterminals, and operators

In 6.005, we will name nonterminals using lowercase identifiers, like `x` or `y` or `url` .

One of the nonterminals of the grammar is designated as the *root*. The set of sentences that the grammar recognizes are the ones that match the root nonterminal. This nonterminal is often called `root` or `start` , but in the grammars below we will typically choose more memorable names like `url` , `html` , and `markdown` .

Grammar Operators

The three most important operators in a production expression are:

- concatenation

`x ::= y z` an `x` is a `y` followed by a `z`

- repetition

`x ::= y*` an `x` is zero or more `y`

- union (also called alternation)

`x ::= y | z` an `x` is a `y` or a `z`

You can also use additional operators which are just syntactic sugar (i.e., they're equivalent to combinations of the big three operators):

- option (0 or 1 occurrence)

```
x ::= y?      an x is a y or is the empty sentence
```

- 1+ repetition (1 or more occurrences)

```
x ::= y+      an x is one or more y
               (equivalent to x ::= y y*)
```

- character classes

```
x ::= [abc]  is equivalent to x ::= 'a' | 'b' | 'c'
```

```
x ::= [^b]    is equivalent to x ::= 'a' | 'c' | 'd' | 'e' | 'f'
               | ... (all other characters)
```

By convention, the operators `*`, `?`, and `+` have highest precedence, which means they are applied first. Alternation `|` has lowest precedence, which means it is applied last. Parentheses can be used to override this precedence, so that a sequence or alternation can be repeated:

- grouping using parentheses

```
x ::= (y z | a b)*  an x is zero or more y-z or a-b pairs
```

Example: URL

Suppose we want to write a grammar that represents URLs. Let's build up a grammar gradually by starting with simple examples and extending the grammar as we go.

Here's a simple URL:

```
http://mit.edu/
```

A grammar that represents the set of sentences containing *only this URL* would look like:

```
url ::= 'http://mit.edu/'
```

But let's generalize it to capture other domains, as well:

```
http://stanford.edu/
http://google.com/
```

We can write this as one line, like this:

```
url ::= 'http://' [a-z]+ '.' [a-z]+ '/'
```

url

|

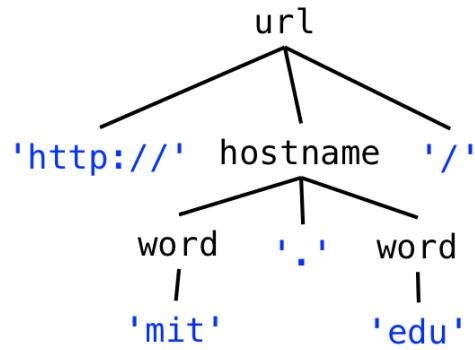
'http://mit.edu/'

This grammar represents the set of all URLs that consist of just a two-part hostname, where each part of the hostname consists of 1 or more letters. So `http://mit.edu/` and `http://yahoo.com/` would match, but not `http://ou812.com/`. Since it has only one nonterminal, a *parse tree* for this URL grammar would look like the picture on the right.

In this one-line form, with a single nonterminal whose production uses only operators and terminals, a grammar is called a *regular expression* (more about that later). But it will be easier to understand if we name the parts using new nonterminals:

```
url ::= 'http://' hostname '/'
hostname ::= word '.' word
word ::= [a-z]+
```

The parse tree for this grammar is now shown at right. The tree has more structure now. The leaves of the tree are the parts of the string that have been parsed. If we concatenated the leaves together, we would recover the original string. The `hostname` and `word` nonterminals are labeling nodes of the tree whose subtrees match those rules in the grammar. Notice that the immediate children of a nonterminal node like `hostname` follow the pattern of the `hostname` rule, `word '.' word`.



How else do we need to generalize? Hostnames can have more than two components, and there can be an optional port number:

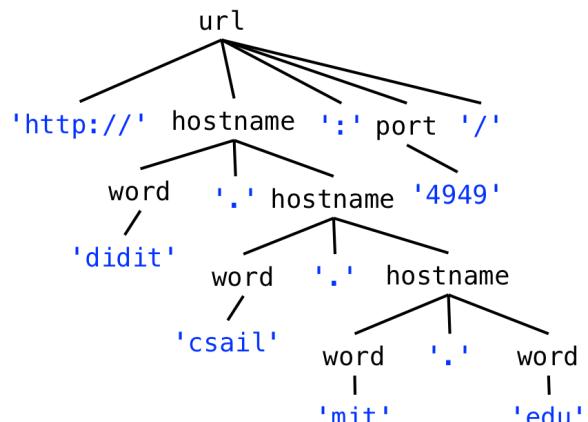
```
http://diddit.csail.mit.edu:4949/
```

To handle this kind of string, the grammar is now:

```
url ::= 'http://' hostname (':' port)?
      '/'

hostname ::= word '.' hostname | word '.'
word
port ::= [0-9]+
word ::= [a-z]+
```

Notice how `hostname` is now defined recursively in terms of itself. Which part of the `hostname` definition is the base case, and which part is the recursive step? What kinds of hostnames are allowed?



Using the repetition operator, we could also write `hostname` like this:

```
hostname ::= (word '.')+ word
```

Another thing to observe is that this grammar allows port numbers that are not technically legal, since port numbers can only range from 0 to 65535. We could write a more complex definition of `port` that would allow only these integers, but that's not typically done in a grammar. Instead, the constraint `0 <= port <= 65535` would be specified alongside the grammar.

There are more things we should do to go farther:

- generalizing `http` to support the additional protocols that URLs can have
- generalizing the `/` at the end to a slash-separated path
- allowing hostnames with the full set of legal characters instead of just `a-z`

Example: Markdown and HTML

Now let's look at grammars for some file formats. We'll be using two different markup languages that represent typographic style in text. Here they are:

Markdown

This is italic.

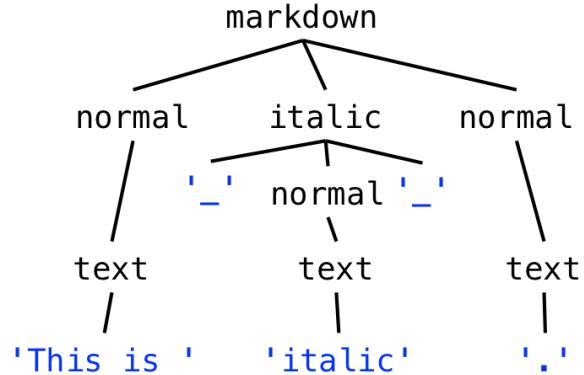
HTML

Here is an *italic* word.

For simplicity, our example HTML and Markdown grammars will only specify italics, but other text styles are of course possible.

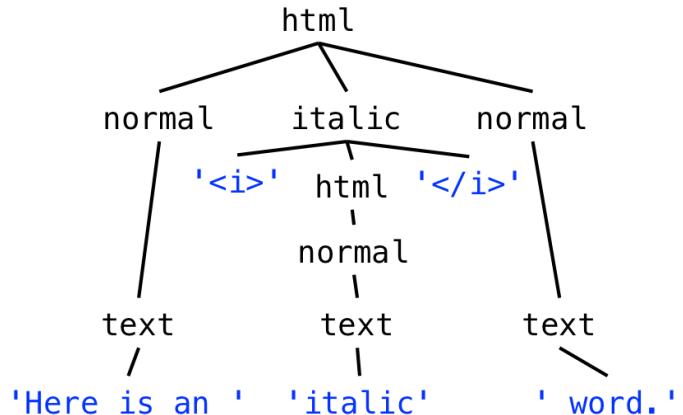
Here's the grammar for our simplified version of Markdown:

```
markdown ::= ( normal | italic ) *
italic ::= '_' normal '_'
normal ::= text
text ::= [^_]*
```



Here's the grammar for our simplified version of HTML:

```
html ::= ( normal | italic ) *
italic ::= '<i>' html '</i>'
normal ::= text
text ::= [^<>]*
```



Regular Expressions

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.

Our URL grammar was regular. By replacing nonterminals with their productions, it can be reduced to a single expression:

```
url ::= 'http://' ([a-z]+ '.')+ [a-z]+ (':' [0-9]+)? '/'
```

The Markdown grammar is also regular:

```
markdown ::= ([^_]*)|(''_')([^_]*)'_')
```

But our HTML grammar can't be reduced completely. By substituting righthand sides for nonterminals, you can eventually reduce it to something like this:

```
html ::= ( [^<>]* | '<i>' html '</i>' ) *
```

...but the recursive use of `html` on the righthand side can't be eliminated, and can't be simply replaced by a repetition operator either. So the HTML grammar is not regular.

The reduced expression of terminals and operators can be written in an even more compact form, called a *regular expression*. A regular expression does away with the quotes around the terminals, and the spaces between terminals and operators, so that it consists just of terminal characters, parentheses for grouping, and operator characters. For example, the regular expression for our `markdown` format is just

```
([^_*]*|_*[^_*]*_)*
```

Regular expressions are also called *regexes* for short. A regex is far less readable than the original grammar, because it lacks the nonterminal names that documented the meaning of each subexpression. But a regex is fast to implement, and there are libraries in many programming languages that support regular expressions.

The regex syntax commonly implemented in programming language libraries has a few more special operators, in addition to the ones we used above in grammars. Here's are some common useful ones:

- any single character
- `\d` any digit, same as `[0-9]`
- `\s` any whitespace character, including space, tab, newline
- `\w` any word character, including letters and digits
- `\., \(, \)`, `*, \+, ...`
escapes an operator or special character so that it matches literally

Using backslashes is important whenever there are terminal characters that would be confused with special characters. Because our `url` regular expression has `.` in it as a terminal, we need to use a backslash to escape it:

```
http://([a-z]+\.)+[a-z]+(:[0-9]+)/*
```

Using regular expressions in Java

Regular expressions (“regexes”) are widely used in programming, and you should have them in your toolbox.

In Java, you can use regexes for manipulating strings (see `String.split` ([, `String.matches` \(\[, `java.util.regex.Pattern` \\(<https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html>\\)\\). They're built-in as a first-class feature of modern scripting languages like Python, Ruby, and Javascript, and you can use them in many text editors for find and replace. Regular expressions are your friend! Most of the time. Here are some examples.\]\(https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#matches-java.lang.String-\)](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#split-java.lang.String-)

Replace all runs of spaces with a single space:

```
String singleSpacedString = string.replaceAll(" +", " ");
```

Match a URL:

```
Pattern regex = Pattern.compile("http://([a-z]+\.\.)+[a-z]+(:[0-9]+)?/");  
Matcher m = regex.matcher(string);  
if (m.matches()) {  
    // then string is a url  
}
```

Extract part of an HTML tag:

```
Pattern regex = Pattern.compile("<a href=['\"]([^\"]*)['\"]>");  
Matcher m = regex.matcher(string);  
if (m.matches()) {  
    String url = m.group(1);  
    // Matcher.group(n) returns the nth parenthesized part of the regex  
}
```

Notice the backslashes in the URL and HTML tag examples. In the URL example, we want to match a literal period . , so we have to first escape it as \. to protect it from being interpreted as the regex match-any-character operator, and then we have to further escape it as \\. to protect the backslash from being interpreted as a Java string escape character. In the HTML example, we have to escape the quote mark " as \' to keep it from ending the string. The frequency of backslash escapes makes regexes still less readable.

Context-Free Grammars

In general, a language that can be expressed with our system of grammars is called context-free. Not all context-free languages are also regular; that is, some grammars can't be reduced to single nonrecursive productions. Our HTML grammar is context-free but not regular.

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 but not regular. That description applies to the Java grammar, shown here in part:

```
statement ::=  
{' statement* '}'  
| 'if' '(' expression ')' statement ('else' statement)?  
| 'for' '(' forinit? ';' expression? ';' forupdate? ')' statement  
| 'while' '(' expression ')' statement  
| 'do' statement 'while' '(' expression ')' ';' '  
| 'try' '{}' statement* '{}' ( catches | catches? 'finally' '{}' statement* '{}')  
| 'switch' '(' expression ')' '{}' switchgroups '{}'  
| 'synchronized' '(' expression ')' '{}' statement* '{}'  
| 'return' expression? ';' '  
| 'throw' expression ';' '  
| 'break' identifier? ';' '  
| 'continue' identifier? ';' '  
| expression ';' '  
| identifier ':' statement  
| ;'
```

Summary

Machine-processed textual languages are ubiquitous in computer science. Grammars are the most popular formalism for describing such languages, and regular expressions are an important subclass of grammars that can be expressed without recursion.

The topics of today's reading connect to our three properties of good software as follows:

- **Safe from bugs.** Grammars and regular expressions are declarative specifications for strings and streams, which can be used directly by libraries and tools. These specifications are often simpler, more direct, and less likely to be buggy than parsing code written by hand.
- **Easy to understand.** A grammar captures the shape of a sequence in a form that is easier to understand than hand-written parsing code. Regular expressions, alas, are often not easy to understand, because they are a one-line reduced form of what might have been a more understandable regular grammar.
- **Ready for change.** A grammar can be easily edited, but regular expressions, unfortunately, are much harder to change, because a complex regular expression is cryptic and hard to understand.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 18: Parser Generators

Parser Generators

An Antlr Grammar

Generating the parser

Calling the parser

Traversing the parse tree

Constructing an abstract syntax tree

Handling errors

Summary

Reading 18: Parser Generators

Software in 6.005

Main.java line 521 (<https://github.com/mit-6.005/s16/tree/18-parser-generators/blob/master/src/intexpr/Main.java#L521-L201>)

~~Safe from bugs~~

~~Easy to understand~~

Ready for change

Correct today and correct in the unknown future.

Communicating clearly with future programmers, including future you.

Designed to accommodate change without rewriting.

Objectives

After today's class, you should:

- Be able to use a grammar in combination with a parser generator, to parse a character sequence into a parse tree
- Be able to convert a parse tree into a useful data type

Parser Generators

A *parser generator* is a good tool that you should make part of your toolbox. A parser generator takes a grammar as input and automatically generates source code that can parse streams of characters using the grammar.

The generated code is a *parser*, which takes a sequence of characters and tries to match the sequence against the grammar. The parser typically produces a *parse tree*, which shows how grammar productions are expanded into a sentence that matches the character sequence. The root of the parse tree is the starting nonterminal of the grammar. Each node of the parse tree expands into one production of the grammar. We'll see how a parse tree actually looks in the next section.

The final step of parsing is to do something useful with this parse tree. We're going to translate it into a value of a recursive data type. Recursive abstract data types are often used to represent an expression in a language, like HTML, or Markdown, or Java, or algebraic expressions. A recursive abstract data type that represents a language expression is called an *abstract syntax tree* (AST).

Antlr is a mature and widely-used parser generator for Java, and other languages as well. The remainder of this reading will get you started with Antlr. If you run into trouble and need a deeper reference, you can look at:

- Definitive Antlr 4 Reference (<https://www.google.com/search?q=The+Definitive+Antlr+4+Reference>) . A book about Antlr, both tutorial and reference.
- Antlr 4 Documentation Wiki (<https://theantlrguy.atlassian.net/wiki/display/ANTLR4/ANTLR+4+Documentation>) . Concise documentation of the grammar file syntax.
- Antlr 4 Runtime API ([//www.antlr.org/api/Java/](https://www.antlr.org/api/Java/)) . Reference documentation for Antlr's Java classes and interfaces.

An Antlr Grammar

The code for the examples that follow can be found on GitHub as **sp16-ex18-parser-generators** (<https://github.com/mit6005/sp16-ex18-parser-generators>) .

Here is what our HTML grammar looks like as an Antlr source file:

```
grammar Html;

root : html EOF;
html : ( italic | normal ) *;
italic : '<i>' html '</i>';
normal : TEXT;
TEXT : ~[<>]+; /* represents a string of one or more characters that are not < or >
```

Let's break it down.

Each Antlr rule consists of a name, followed by a colon, followed by its definition, terminated by a semicolon.

Nonterminals in Antlr have to be lowercase: `root` , `html` , `normal` , `italic` . Terminals are either quoted strings, like '`<i>`' , or capitalized names, like `EOF` and `TEXT` .

```
root : html EOF;
```

`root` is the entry point of the grammar. This is the nonterminal that the whole input needs to match. We don't have to call it `root` . The entry point can be any nonterminal.

`EOF` is a special terminal, defined by Antlr, that means the end of the input. It stands for *end of file* , though your input may also come from a string or a network connection rather than just a file.

```
html : ( normal | italic ) *;
```

This rule shows that Antlr rules can have the alternation operator `|` , the repetition operators `*` and `+` , and parentheses for grouping, in the same way we've been using in the grammars reading [\(../17-regex-grammars/#grammars\)](#) . Optional parts can be marked with `?` , just like we did earlier, but this particular grammar doesn't use `?` .

```
italic : '<i>' html '</i>';
normal : TEXT;
TEXT : ~[<>]+;
```

`TEXT` is a terminal matching sequences of characters that are neither `<` nor `>`. In the more conventional regular expression syntax used earlier in this reading, we would write `[^<>]` to represent all characters except `<` and `>`. Antlr uses a slightly different syntax – `~` means *not*, and it is put in front of the square brackets instead of inside them, so `~[<>]` matches any character except `<` and `>`.

In Antlr, terminals can be defined using regular expressions, not just fixed strings. For example, here are some other terminal patterns we used in the URL grammar earlier in the reading, now written in Antlr syntax and with Antlr's required naming convention:

```
IDENTIFIER : [a-z]+;
INTEGER : [0-9]+;
```

More about Antlr's grammar file syntax can be found in Chapter 5 of the Definitive ANTLR 4 Reference (<https://www.google.com/search?q=The+Definitive+Antlr+4+Reference>) .

Generating the parser

The rest of this reading will focus on the `IntegerExpression` grammar used in the exercise above, which we'll store in a file called `IntegerExpression.g4` (<https://github.com/mit6005/sp16-ex18-parser-generators/blob/master/src/intexpr/parser/IntegerExpression.g4>). Antlr 4 grammar files end with `.g4` by convention.

The Antlr parser generator tool converts a grammar source file like `IntegerExpression.g4` into Java classes that implement a parser. To do that, you need to go to a command prompt (Terminal or Command Prompt) and run a command like this:

```
cd <root of project>
cd src/intexpr/parser
java -jar ../../lib/antlr.jar IntegerExpression.g4
```

You need to make sure you `cd` into right folder (where `IntegerExpression.g4` is) and you refer to `antlr.jar` where it is in your project folder structure, using a relative path like `../../lib/antlr.jar`).

Assuming you don't have any syntax errors in your grammar file, the parser generator will produce new Java source files in the current folder. Nothing will be printed in the terminal. The generated code is divided into several cooperating modules:

- the **lexer** takes a stream of characters as input, and produces a stream of terminals (Antlr calls them *tokens*) as output, like `NUMBER` , `+` , and `(` . For `IntegerExpression.g4` , the generated lexer is called `IntegerExpressionLexer.java` .
- the **parser** takes the stream of terminals produced by the lexer and produces a parse tree. The generated parser is called `IntegerExpressionParser.java` .
- the **tree walker** lets you write code that walks over the parse tree produced by the parser, as explained below. The generated tree walker files are the interface `IntegerExpressionListener.java` , and an empty implementation of the interface, `IntegerExpressionBaseListener.java` .

Antlr also generates two text files, `IntegerExpression.tokens` and `IntegerExpressionLexer.tokens`, that list the terminals that Antlr found in your grammar. These aren't needed for a simple parser, but they're needed when you include grammars inside other grammars.

Make sure that you:

- **Never edit the files generated by Antlr.** The right way to change your parser is to edit the grammar source file, `IntegerExpression.g4`, and then regenerate the Java classes.
- **Regenerate the files whenever you edit the grammar file.** This is easy to forget when Eclipse is compiling all your Java source files automatically. Eclipse does not regenerate your parser automatically. Make sure you rerun the `java -jar ...` command whenever you change your `.g4` file.
- **Refresh your project in Eclipse each time you regenerate the files.** You can do this by clicking on your project in Eclipse and pressing F5, or right-clicking and choosing Refresh. The reason is that Eclipse sometimes uses older versions of files already part of the source, even if they have been modified on your filesystem.

More about using the Antlr parser generator to produce a parser can be found in Chapter 3 of the Definitive ANTLR 4 Reference (<https://www.google.com/search?q=The+Definitive+Antlr+4+Reference>) .

Calling the parser

Now that you've generated the Java classes for your parser, you'll want to use them from your own code.

First we need to make a stream of characters to feed to the lexer. Antlr has a class `ANTLRInputStream` that makes this easy. It can take a `String`, or a `Reader`, or an `InputStream` as input. Here we are using a string:

```
CharStream stream = new ANTLRInputStream("54+(2+89)");
```

Next, we create an instance of the lexer class that our grammar file generated, and pass it the character stream:

```
IntegerExpressionLexer lexer = new IntegerExpressionLexer(stream);
TokenStream tokens = new CommonTokenStream(lexer);
```

The result is a stream of terminals, which we can then feed to the parser:

```
IntegerExpressionParser parser = new IntegerExpressionParser(tokens);
```

To actually do the parsing, we call a particular nonterminal on the parser. The generated parser has one method for every nonterminal in our grammar, including `root()`, `sum()`, and `primitive()`. We want to call the nonterminal that represents the set of strings that we want to match – in this case, `root()`.

Calling it produces a parse tree:

```
ParseTree tree = parser.root();
```

For debugging, we can then print this tree out:

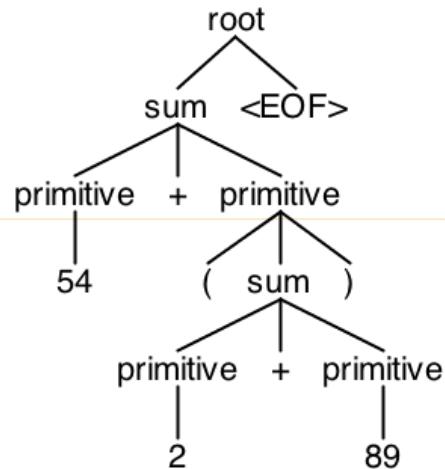
```
System.out.println(tree.toStringTree(parser));
```

Or we can display it in a handy graphical form:

```
Trees.inspect(tree, parser);
```

which pops up a window with the parse tree shown on the right.

In the example code: `Main.java` lines 34-50 (<https://github.com/mit6005/sp16-ex18-parser-generators/blob/master/src/intexpr/Main.java#L34-L50>) , which use lines 71-85 (<https://github.com/mit6005/sp16-ex18-parser-generators/blob/master/src/intexpr/Main.java#L71-L85>) .



Traversing the parse tree

So we've used the parser to turn a stream of characters into a parse tree, which shows how the grammar matches the stream. Now we need to do something with this parse tree. We're going to translate it into a value of a recursive abstract data type.

The first step is to learn how to traverse the parse tree. To do this, we use a `ParseTreeWalker` , which is an Antlr class that walks over a parse tree, visiting every node in order, top-to-bottom, left-to-right. As it visits each node in the tree, the walker calls methods on a *listener* object that we provide, which implements `IntegerExpressionListener` interface.

Just to warm up, here's a simple implementation of `IntegerExpressionListener` that just prints a message every time the walker calls us, so we can see how it gets used:

```

class PrintEverything implements IntegerExpressionListener {

    @Override public void enterRoot(IntegerExpressionParser.RootContext context)
    {
        System.out.println("entering root");
    }
    @Override public void exitRoot(IntegerExpressionParser.RootContext context) {
        System.out.println("exiting root");
    }

    @Override public void enterSum(IntegerExpressionParser.SumContext context) {
        System.out.println("entering sum");
    }
    @Override public void exitSum(IntegerExpressionParser.SumContext context) {
        System.out.println("exiting sum");
    }

    @Override public void enterPrimitive(IntegerExpressionParser.PrimitiveContext
context) {
        System.out.println("entering primitive");
    }
    @Override public void exitPrimitive(IntegerExpressionParser.PrimitiveContext
context) {
        System.out.println("exiting primitive");
    }

    @Override public void visitTerminal(TerminalNode terminal) {
        System.out.println("terminal " + terminal.getText());
    }

    // don't need these here, so just make them empty implementations
    @Override public void enterEveryRule(ParserRuleContext context) { }
    @Override public void exitEveryRule(ParserRuleContext context) { }
    @Override public void visitErrorNode(ErrorNode node) { }
}

```

Notice that every nonterminal N in the grammar has corresponding `enterN()` and `exitN()` methods in the listener interface, which are called when the tree walk enters and exits a parse tree node for nonterminal N , respectively. There is also a `visitTerminal()` that is called when the walk reaches a leaf of the parse tree. Each of these methods has a parameter that provides information about the nonterminal or terminal node that the walk is currently visiting.

The listener interface also has some methods that we don't need. The methods `enterEveryRule()` and `exitEveryRule()` are called on entering and exiting *any* nonterminal node, in case we want some generic behavior. The method `visitErrorNode()` is called if the input contained a syntax error that produced an error node in the parse tree. In the parser we're writing, however, a syntax error causes an exception to be thrown, so we won't see any parse trees with error nodes in them. The interface requires us to implement these methods, but we can just leave their method bodies empty.

```

ParseTreeWalker walker = new ParseTreeWalker();
IntegerExpressionListener listener = new PrintEverything();
walker.walk(listener, tree);

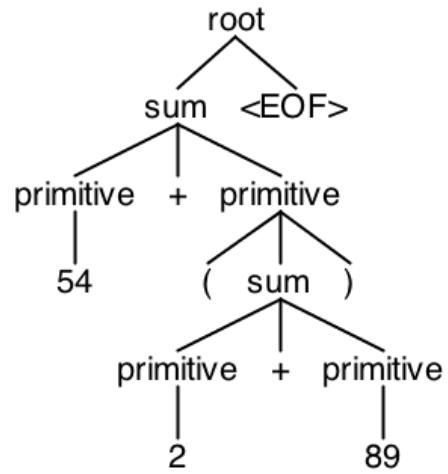
```

If we walk over the parse tree with this listener, then we see the following output:

```

entering root
entering sum
entering primitive
terminal 54
exiting primitive
terminal +
entering primitive
terminal (
entering sum
entering primitive
terminal 2
exiting primitive
terminal +
entering primitive
terminal 89
exiting primitive
exit sum
terminal )
exiting primitive
exit sum
terminal <EOF>
exiting root

```



Compare this printout with the parse tree shown at the right, and you'll see that the `ParseTreeWalker` is stepping through the nodes of the tree in order, from parents to children, and from left to right through the siblings.

Constructing an abstract syntax tree

We need to convert the parse tree into a recursive data type. Here's the definition of the recursive data type that we're going to use to represent integer arithmetic expressions:

```

IntegerExpression = Number(n:int)
                  + Plus(left:IntegerExpression, right:IntegerExpression)

```

If this syntax is mysterious, review recursive data type definitions ([../16-recursive-data-types/recursive/#recursive_datatype_definitions](#)) .

When a recursive data type represents a language this way, it is often called an *abstract syntax tree* . A `IntegerExpression` (<https://github.com/mit6005/sp16-ex18-parser-generators/blob/master/src/intexpr/IntegerExpression.java>) value captures the important features of the expression – its grouping and the integers in it – while omitting unnecessary details of the sequence of characters that created it.

By contrast, the parse tree that we just generated with the `IntegerExpression` parser is a *concrete syntax tree* . It's called concrete, rather than abstract, because it contains more details about how the expression is represented in actual characters. For example, the strings `2+2` , `((2)+(2))` , and `0002+0002` would each produce a different concrete syntax tree, but these trees would all correspond to the same abstract `IntegerExpression` value: `Plus(Number(2), Number(2))` .

Now we create a listener that constructs a `IntegerExpression` tree while it's walking over the parse tree. Each parse tree node will correspond to a `IntegerExpression` variant: `sum` nodes will create `Plus` (<https://github.com/mit6005/sp16-ex18-parser-generators/blob/master/src/intexpr/IntegerExpression.java>)

generators/blob/master/src/intexpr/IntegerExpression.java#L36) objects, and primitive nodes (that matched the NUMBER terminal) will create Number (<https://github.com/mit6005/sp16-ex18-parser-generators/blob/master/src/intexpr/IntegerExpression.java#L12>) objects.

Some primitive nodes are parenthesized subexpressions, not numbers. For these nodes, our listener will construct no new IntegerExpression object at all. Parentheses are concrete syntax whose meaning is captured in the abstract syntax tree by the structure of Plus and Number objects, and we already have a Plus to represent the sum inside the parentheses.

Whenever the walker exits each node of the parse tree, we have walked over the entire subtree under that node, so we create the next IntegerExpression object at exit time. But we have to keep track of all the children that were created during the walk over that subtree. We use a stack to store them.

Here's the code:

```

/** Make a IntegerExpression value from a parse tree. */
class MakeIntegerExpression implements IntegerExpressionListener {
    private Stack<IntegerExpression> stack = new Stack<>();
    // Invariant: stack contains the IntegerExpression value of each parse
    // subtree that has been fully-walked so far, but whose parent has not yet
    // been exited by the walk. The stack is ordered by recency of visit, so that
    // the top of the stack is the IntegerExpression for the most recently walked
    // subtree.
    //
    // At the start of the walk, the stack is empty, because no subtrees have
    // been fully walked.
    //
    // Whenever a node is exited by the walk, the IntegerExpression values of its
    // children are on top of the stack, in order with the last child on top. To
    // preserve the invariant, we must pop those child IntegerExpression values
    // from the stack, combine them with the appropriate IntegerExpression
    // producer, and push back an IntegerExpression value representing the entire
    // subtree under the node.
    //
    // At the end of the walk, after all subtrees have been walked and the root
    // has been exited, only the entire tree satisfies the invariant's
    // "fully walked but parent not yet exited" property, so the top of the stack
    // is the IntegerExpression of the entire parse tree.

    /**
     * Returns the expression constructed by this listener object.
     * Requires that this listener has completely walked over an IntegerExpression
     */
    public IntegerExpression getExpression() {
        return stack.get(0);
    }

    @Override public void exitRoot(IntegerExpressionParser.RootContext context) {
        // do nothing, root has only one child so its value is
        // already on top of the stack
    }

    @Override public void exitSum(IntegerExpressionParser.SumContext context) {
        // matched the primitive ('+' primitive)* rule
        List<IntegerExpressionParser.PrimitiveContext> addends = context.primitive();
        assert stack.size() >= addends.size();

        // the pattern above always has at least 1 child;
        // pop the last child
        assert addends.size() > 0;
        IntegerExpression sum = stack.pop();

        // pop the older children, one by one, and add them on
        for (int i = 1; i < addends.size(); ++i) {
            sum = new Plus(stack.pop(), sum);
        }
    }
}

```

```

    // the result is this subtree's IntegerExpression
    stack.push(sum);
}

@Override public void exitPrimitive(IntegerExpressionParser.PrimitiveContext
context) {
    if (context.NUMBER() != null) {
        // matched the NUMBER alternative
        int n = Integer.valueOf(context.NUMBER().getText());
        IntegerExpression number = new Number(n);
        stack.push(number);
    } else {
        // matched the '(' sum ')' alternative
        // do nothing, because sum's value is already on the stack
    }
}

// don't need these here, so just make them empty implementations
@Override public void enterRoot(IntegerExpressionParser.RootContext context)
{ }
@Override public void enterSum(IntegerExpressionParser.SumContext context) {
}
@Override public void enterPrimitive(IntegerExpressionParser.PrimitiveContext
context) { }

@Override public void visitTerminal(TerminalNode terminal) { }
@Override public void enterEveryRule(ParserRuleContext context) { }
@Override public void exitEveryRule(ParserRuleContext context) { }
@Override public void visitErrorNode(ErrorNode node) { }
}

```

More about Antlr's parse-tree listeners can be found in Section 7.2 of the Definitive ANTLR 4 Reference (<https://www.google.com/search?q=The+Definitive+Antlr+4+Reference>) .

Handling errors

By default, Antlr parsers print errors to the console. In order to make the parser modular, however, we need to handle those errors differently. You can attach an `ErrorListener` to the lexer and parser in order to throw an exception when an error is encountered during parsing. The `Configuration.g4` file defines a method `reportErrorsAsExceptions()` which does this. So if you copy the technique used in this grammar file, you can call:

```

lexer.reportErrorsAsExceptions();
parser.reportErrorsAsExceptions();

```

right after you create the lexer and parser. Then when you call `parser.root()`, it will throw an exception as soon as it encounters something that it can't match.

This is a simplistic approach to handling errors. Antlr offers more sophisticated forms of error recovery as well. To learn more, see Chapter 9 in the Definitive Antlr 4 Reference (<https://www.google.com/search?q=The+Definitive+Antlr+4+Reference>) .

Summary

The topics of today's reading connect to our three properties of good software as follows:

- **Safe from bugs.** A grammar is a declarative specification for strings and streams, which can be implemented automatically by a parser generator. These specifications are often simpler, more direct, and less likely to be buggy than parsing code written by hand.
- **Easy to understand.** A grammar captures the shape of a sequence in a form that is compact and easier to understand than hand-written parsing code.
- **Ready for change.** A grammar can be easily edited, then run through a parser generator to regenerate the parsing code.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 19: Concurrency

Concurrency

Two Models for Concurrent Programming

Processes, Threads, Time-slicing

Shared Memory Example

Interleaving

Race Condition

Tweaking the Code Won't Help

Reordering

Message Passing Example

Concurrency is Hard to Test and Debug

Summary

Reading 19: Concurrency

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- The message passing and shared memory models of concurrency
- Concurrent processes and threads, and time slicing
- The danger of race conditions

Concurrency

Concurrency means multiple computations are happening at the same time. Concurrency is everywhere in modern programming, whether we like it or not:

- Multiple computers in a network
- Multiple applications running on one computer
- Multiple processors in a computer (today, often multiple processor cores on a single chip)

In fact, concurrency is essential in modern programming:

- Web sites must handle multiple simultaneous users.

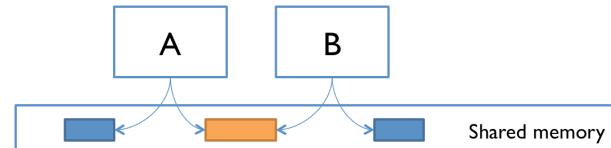
- Mobile apps need to do some of their processing on servers (“in the cloud”).
- Graphical user interfaces almost always require background work that does not interrupt the user. For example, Eclipse compiles your Java code while you’re still editing it.

Being able to program with concurrency will still be important in the future. Processor clock speeds are no longer increasing. Instead, we’re getting more cores with each new generation of chips. So in the future, in order to get a computation to run faster, we’ll have to split up a computation into concurrent pieces.

Two Models for Concurrent Programming

There are two common models for concurrent programming: *shared memory* and *message passing*.

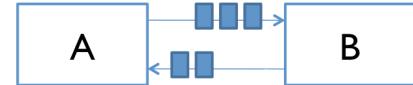
Shared memory. In the shared memory model of concurrency, concurrent modules interact by reading and writing shared objects in memory.



Examples of the shared-memory model:

- A and B might be two processors (or processor cores) in the same computer, sharing the same physical memory.
- A and B might be two programs running on the same computer, sharing a common filesystem with files they can read and write.
- A and B might be two threads in the same Java program (we’ll explain what a thread is below), sharing the same Java objects.

Message passing. In the message-passing model, concurrent modules interact by sending messages to each other through a communication channel. Modules send off messages, and incoming messages to each module are queued up for handling. Examples include:



- A and B might be two computers in a network, communicating by network connections.
- A and B might be a web browser and a web server – A opens a connection to B and asks for a web page, and B sends the web page data back to A.
- A and B might be an instant messaging client and server.
- A and B might be two programs running on the same computer whose input and output have been connected by a pipe, like `ls | grep` typed into a command prompt.

Processes, Threads, Time-slicing

The message-passing and shared-memory models are about how concurrent modules communicate. The concurrent modules themselves come in two different kinds: processes and threads.

Process. A process is an instance of a running program that is *isolated* from other processes on the same machine. In particular, it has its own private section of the machine’s memory.

The process abstraction is a *virtual computer*. It makes the program feel like it has the entire machine to itself – like a fresh computer has been created, with fresh memory, just to run that program.

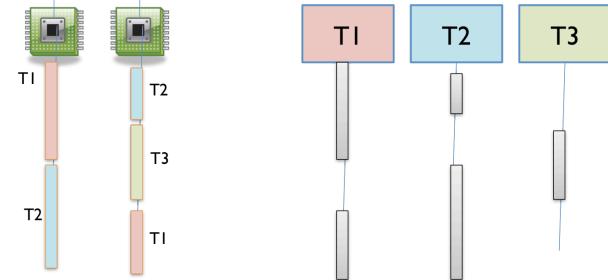
Just like computers connected across a network, processes normally share no memory between them. A process can’t access another process’s memory or objects at all. Sharing memory between processes is *possible* on most operating systems, but it needs special effort. By contrast, a new process is automatically ready for message passing, because it is created with standard input & output streams, which are the `System.out` and `System.in` streams you’ve used in Java.

Thread. A thread is a locus of control inside a running program. Think of it as a place in the program that is being run, plus the stack of method calls that led to that place (so the thread can go back up the stack when it reaches `return` statements).

Just as a process represents a virtual computer, the thread abstraction represents a *virtual processor*. Making a new thread simulates making a fresh processor inside the virtual computer represented by the process. This new virtual processor runs the same program and shares the same memory as other threads in the process.

Threads are automatically ready for shared memory, because threads share all the memory in the process. It takes special effort to get “thread-local” memory that’s private to a single thread. It’s also necessary to set up message-passing explicitly, by creating and using queue data structures. We’ll talk about how to do that in a future reading.

How can I have many concurrent threads with only one or two processors in my computer? When there are more threads than processors, concurrency is simulated by **time slicing**, which means that the processor switches between threads. The figure on the right shows how three threads T1, T2, and T3 might be time-sliced on a machine that has only two actual processors. In the figure, time proceeds downward, so at first one processor is running thread T1 and the other is running thread T2, and then the second processor switches to run thread T3. Thread T2 simply pauses, until its next time slice on the same processor or another processor.



On most systems, time slicing happens unpredictably and nondeterministically, meaning that a thread may be paused or resumed at any time.

In the Java Tutorials, read:

- **Processes & Threads**
[\(just 1 page\)](https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html)
- **Defining and Starting a Thread**
[\(just 1 page\)](https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html)

The second Java Tutorials reading shows two ways to create a thread.

- Never use their second way (subclassed `Thread`).
- Always implement the `Runnable` [interface](https://docs.oracle.com/javase/8/docs/api/?java/lang/Runnable.html) and use the `new Thread(...)` constructor.

Their example declares a named class that implements `Runnable`:

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
}
// ... in the main method:
new Thread(new HelloRunnable()).start();
```

A very common idiom is starting a thread with an anonymous [Runnable](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html), which eliminates the named class:

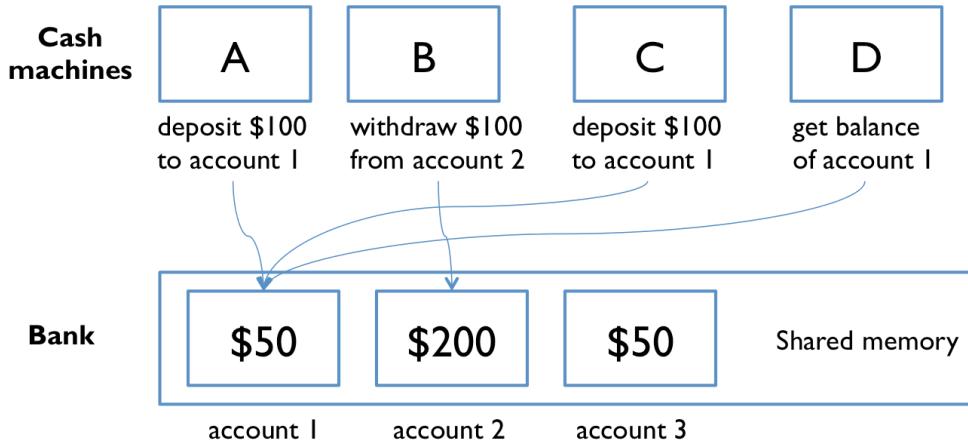
```
new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello from a thread!");
    }
}).start();
```

Read: **using an anonymous Runnable to start a thread (anonymous-runnable/)**

Shared Memory Example

Let's look at an example of a shared memory system. The point of this example is to show that concurrent programming is hard, because it can have subtle bugs.

Imagine that a bank has cash machines that use a shared memory model, so all the cash machines can read and write the same account objects in memory.



To illustrate what can go wrong, let's simplify the bank down to a single account, with a dollar balance stored in the `balance` variable, and two operations `deposit` and `withdraw` that simply add or remove a dollar:

```
// suppose all the cash machines share a single bank account
private static int balance = 0;

private static void deposit() {
    balance = balance + 1;
}
private static void withdraw() {
    balance = balance - 1;
}
```

Customers use the cash machines to do transactions like this:

```
deposit(); // put a dollar in
withdraw(); // take it back out
```

In this simple example, every transaction is just a one dollar deposit followed by a one-dollar withdrawal, so it should leave the balance in the account unchanged. Throughout the day, each cash machine in our network is processing a sequence of deposit/withdraw transactions.

```
// each ATM does a bunch of transactions that
// modify balance, but leave it unchanged afterward
private static void cashMachine() {
    for (int i = 0; i < TRANSACTIONS_PER_MACHINE; ++i) {
        deposit(); // put a dollar in
        withdraw(); // take it back out
    }
}
```

So at the end of the day, regardless of how many cash machines were running, or how many transactions we processed, we should expect the account balance to still be 0.

But if we run this code, we discover frequently that the balance at the end of the day is *not* 0. If more than one `cashMachine()` call is running at the same time – say, on separate processors in the same computer – then `balance` may not be zero at the end of the day. Why not?

Interleaving

Here's one thing that can happen. Suppose two cash machines, A and B, are both working on a deposit at the same time. Here's how the `deposit()` step typically breaks down into low-level processor instructions:

get balance (balance=0)

add 1

write back the result (balance=1)

When A and B are running concurrently, these low-level instructions interleave with each other (some might even be simultaneous in some sense, but let's just worry about interleaving for now):

A	B
A get balance (balance=0)	
A add 1	
A write back the result (balance=1)	
	B get balance (balance=1)
	B add 1
	B write back the result (balance=2)

This interleaving is fine – we end up with balance 2, so both A and B successfully put in a dollar. But what if the interleaving looked like this:

A	B
A get balance (balance=0)	
	B get balance (balance=0)
A add 1	
	B add 1

A**B**

A write back the result (balance=1)

B write back the result (balance=1)

The balance is now 1 – A’s dollar was lost! A and B both read the balance at the same time, computed separate final balances, and then raced to store back the new balance – which failed to take the other’s deposit into account.

Race Condition

This is an example of a **race condition**. A race condition means that the correctness of the program (the satisfaction of postconditions and invariants) depends on the relative timing of events in concurrent computations A and B. When this happens, we say “A is in a race with B.”

Some interleavings of events may be OK, in the sense that they are consistent with what a single, nonconcurrent process would produce, but other interleavings produce wrong answers – violating postconditions or invariants.

Tweaking the Code Won’t Help

All these versions of the bank-account code exhibit the same race condition:

```
// version 1
private static void deposit() { balance = balance + 1; }
private static void withdraw() { balance = balance - 1; }

// version 2
private static void deposit() { balance += 1; }
private static void withdraw() { balance -= 1; }

// version 3
private static void deposit() { ++balance; }
private static void withdraw() { --balance; }
```

You can’t tell just from looking at Java code how the processor is going to execute it. You can’t tell what the indivisible operations – the atomic operations – will be. It isn’t atomic just because it’s one line of Java. It doesn’t touch balance only once just because the balance identifier occurs only once in the line. The Java compiler, and in fact the processor itself, makes no commitments about what low-level operations it will generate from your code. In fact, a typical modern Java compiler produces exactly the same code for all three of these versions!

The key lesson is that you can’t tell by looking at an expression whether it will be safe from race conditions.

Read: **Thread Interference**

(<https://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html>) (just 1 page)

Reordering

It’s even worse than that, in fact. The race condition on the bank account balance can be explained in terms of different interleavings of sequential operations on different processors. But in fact, when you’re using multiple variables and multiple processors, you can’t even count on changes to those variables

appearing in the same order.

Here's an example. Note that it uses a loop that continuously checks for a concurrent condition; this is called busy waiting (https://en.wikipedia.org/wiki/Busy_waiting) and it is not a good pattern. In this case, the code is also broken:

```
private boolean ready = false;
private int answer = 0;

// computeAnswer runs in one thread
private void computeAnswer() {
    answer = 42;
    ready = true;
}

// useAnswer runs in a different thread
private void useAnswer() {
    while (!ready) {
        Thread.yield();
    }
    if (answer == 0) throw new RuntimeException("answer wasn't ready!");
}
```

We have two methods that are being run in different threads. `computeAnswer` does a long calculation, finally coming up with the answer 42, which it puts in the `answer` variable. Then it sets the `ready` variable to true, in order to signal to the method running in the other thread, `useAnswer`, that the answer is ready for it to use. Looking at the code, `answer` is set before `ready` is set, so once `useAnswer` sees `ready` as true, then it seems reasonable that it can assume that the `answer` will be 42, right? Not so.

The problem is that modern compilers and processors do a lot of things to make the code fast. One of those things is making temporary copies of variables like `answer` and `ready` in faster storage (registers or caches on a processor), and working with them temporarily before eventually storing them back to their official location in memory. The storeback may occur in a different order than the variables were manipulated in your code. Here's what might be going on under the covers (but expressed in Java syntax to make it clear). The processor is effectively creating two temporary variables, `tmp_r` and `tmp_a`, to manipulate the fields `ready` and `answer`:

```
private void computeAnswer() {
    boolean tmp_r = ready;
    int tmp_a = answer;

    tmp_a = 42;
    tmp_r = true;

    ready = tmp_r;
        // <-- what happens if useAnswer() interleaves here?
        // ready is set, but answer isn't.
    answer = tmp_a;
}
```

Message Passing Example

Now let's look at the message-passing approach to our bank account example.

Now not only are the cash machine modules, but the accounts are modules, too. Modules interact by sending messages to each other.

Incoming requests are placed in a queue to be handled one at a time. The sender doesn't stop working while waiting for an

answer to its request. It handles more requests from its own queue. The reply to its request eventually comes back as another message.

Unfortunately, message passing doesn't eliminate the possibility of race conditions. Suppose each account supports `get-balance` and `withdraw` operations, with corresponding messages. Two users, at cash machines A and B, are both trying to withdraw a dollar from the same account. They check the balance first to make sure they never withdraw more than the account holds, because overdrafts trigger big bank penalties:

```
get-balance
if balance >= 1 then withdraw 1
```

The problem is again interleaving, but this time interleaving of the *messages* sent to the bank account, rather than the *instructions* executed by A and B. If the account starts with a dollar in it, then what interleaving of messages will fool A and B into thinking they can both withdraw a dollar, thereby overdrawning the account?

One lesson here is that you need to carefully choose the operations of a message-passing model. `withdraw-if-sufficient-funds` would be a better operation than just `withdraw`.

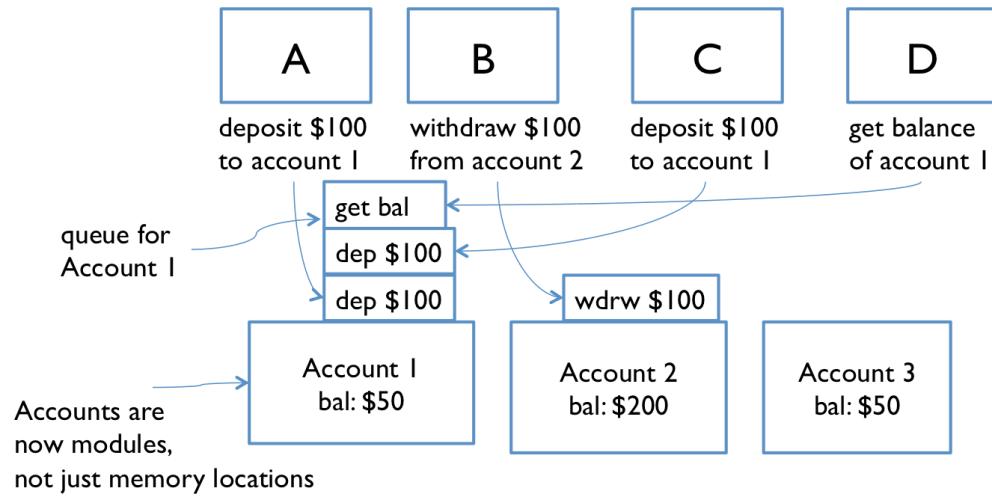
Concurrency is Hard to Test and Debug

If we haven't persuaded you that concurrency is tricky, here's the worst of it. It's very hard to discover race conditions using testing. And even once a test has found a bug, it may be very hard to localize it to the part of the program causing it.

Concurrency bugs exhibit very poor reproducibility. It's hard to make them happen the same way twice. Interleaving of instructions or messages depends on the relative timing of events that are strongly influenced by the environment. Delays can be caused by other running programs, other network traffic, operating system scheduling decisions, variations in processor clock speed, etc. Each time you run a program containing a race condition, you may get different behavior.

These kinds of bugs are *heisenbugs*, which are nondeterministic and hard to reproduce, as opposed to a *bohrbug*, which shows up repeatedly whenever you look at it. Almost all bugs in sequential programming are bohrbugs.

A heisenbug may even disappear when you try to look at it with `println` or debugger! The reason is that printing and debugging are so much slower than other operations, often 100-1000x slower, that they dramatically change the timing of operations, and the interleaving. So inserting a simple print statement into the `cashMachine()`:



```
private static void cashMachine() {  
    for (int i = 0; i < TRANSACTIONS_PER_MACHINE; ++i) {  
        deposit(); // put a dollar in  
        withdraw(); // take it back out  
        System.out.println(balance); // makes the bug disappear!  
    }  
}
```

...and suddenly the balance is always 0, as desired, and the bug appears to disappear. But it's only masked, not truly fixed. A change in timing somewhere else in the program may suddenly make the bug come back.

Concurrency is hard to get right. Part of the point of this reading is to scare you a bit. Over the next several readings, we'll see principled ways to design concurrent programs so that they are safer from these kinds of bugs.

Summary

- Concurrency: multiple computations running simultaneously
- Shared-memory & message-passing paradigms
- Processes & threads
 - Process is like a virtual computer; thread is like a virtual processor
- Race conditions
 - When correctness of result (postconditions and invariants) depends on the relative timing of events

These ideas connect to our three key properties of good software mostly in bad ways. Concurrency is necessary but it causes serious problems for correctness. We'll work on fixing those problems in the next few readings.

- **Safe from bugs.** Concurrency bugs are some of the hardest bugs to find and fix, and require careful design to avoid.
- **Easy to understand.** Predicting how concurrent code might interleave with other concurrent code is very hard for programmers to do. It's best to design your code in such a way that programmers don't have to think about interleaving at all.
- **Ready for change.** Not particularly relevant here.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 20: Thread Safety

What Threadsafe Means

Strategy 1: Confinement

Strategy 2: Immutability

Strategy 3: Using Threadsafe Data Types

How to Make a Safety Argument

Summary

Reading 20: Thread Safety

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

Recall race conditions: multiple threads sharing the same mutable variable without coordinating what they're doing. This is unsafe, because the correctness of the program may depend on accidents of timing of their low-level operations.

There are basically four ways to make variable access safe in shared-memory concurrency:

- **Confinement.** Don't share the variable between threads. This idea is called confinement, and we'll explore it today.
- **Immutability.** Make the shared data immutable. We've talked a lot about immutability already, but there are some additional constraints for concurrent programming that we'll talk about in this reading.
- **Threadsafe data type.** Encapsulate the shared data in an existing threadsafe data type that does the coordination for you. We'll talk about that today.
- **Synchronization.** Use synchronization to keep the threads from accessing the variable at the same time. Synchronization is what you need to build your own threadsafe data type.

We'll talk about the first three ways in this reading, along with how to make an argument that your code is threadsafe using those three ideas. We'll talk about the fourth approach, synchronization, in a later reading.

The material in this reading is inspired by an excellent book: Brian Goetz et al., *Java Concurrency in Practice* ([//jcip.net/](http://jcip.net/)) , Addison-Wesley, 2006.

What Threadsafe Means

A data type or static method is *threadsafe* if it behaves correctly when used from multiple threads, regardless of how those threads are executed, and without demanding additional coordination from the calling code.

- “behaves correctly” means satisfying its specification and preserving its rep invariant;
- “regardless of how threads are executed” means threads might be on multiple processors or timesliced on the same processor;
- “without additional coordination” means that the data type can’t put preconditions on its caller related to timing, like “you can’t call `get()` while `set()` is in progress.”

Remember `Iterator` ([//docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html](https://docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html)) ? It’s not threadsafe. `Iterator`’s specification says that you can’t modify a collection at the same time as you’re iterating over it. That’s a timing-related precondition put on the caller, and `Iterator` makes no guarantee to behave correctly if you violate it.

Strategy 1: Confinement

Our first way of achieving thread safety is *confinement*. Thread confinement is a simple idea: you avoid races on mutable data by keeping that data confined to a single thread. Don’t give any other threads the ability to read or write the data directly.

Since shared mutable data is the root cause of a race condition, confinement solves it by *not sharing* the mutable data.

Local variables are always thread confined. A local variable is stored in the stack, and each thread has its own stack. There may be multiple invocations of a method running at a time (in different threads or even at different levels of a single thread’s stack, if the method is recursive), but each of those invocations has its own private copy of the variable, so the variable itself is confined.

But be careful – the variable is thread confined, but if it’s an object reference, you also need to check the object it points to. If the object is mutable, then we want to check that the object is confined as well – there can’t be references to it that are reachable from any other thread.

Confinement is what makes the accesses to `n`, `i`, and `result` safe in code like this:

```

public class Factorial {

    /**
     * Computes n! and prints it on standard output.
     * @param n must be >= 0
     */
    private static void computeFact(final int n) {
        BigInteger result = new BigInteger("1");
        for (int i = 1; i <= n; ++i) {
            System.out.println("working on fact " + n);
            result = result.multiply(new BigInteger(String.valueOf(i)));
        }
        System.out.println("fact(" + n + ") = " + result);
    }

    public static void main(String[] args) {
        new Thread(new Runnable() { // create a thread using an
            public void run() { // anonymous Runnable
                computeFact(99);
            }
        }).start();
        computeFact(100);
    }
}

```

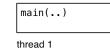
This code starts the thread for `computeFact(99)` with an anonymous [Runnable](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html), a common idiom discussed in the previous reading ([./19-concurrency/anonymous-runnable/](#)).

Let's look at snapshot diagrams for this code. Hover or tap on each step to update the diagram:

- When we start the program, we start with one thread running `main`
- `main` creates a second thread using the anonymous `Runnable` idiom, and starts that thread.
- At this point, we have two concurrent threads of execution. Their interleaving is unknown! But one *possibility* for the next thing that happens is that thread 1 enters `computeFact`.
- Then, the next thing that *might* happen is that thread 2 also enters `computeFact`.

At this point, we see how **confinement** helps with thread safety: each execution of `computeFact` has its own `n`, `i`, and `result` variables. None of the objects they point to are mutable; if they were mutable, we would need to check that the objects are not aliased from other threads.

- The `computeFact` computations proceed independently, updating their respective variables.



Avoid Global Variables

Unlike local variables, static variables are not automatically thread confined.

If you have static variables in your program, then you have to make an argument that only one thread will ever use them, and you have to document that fact clearly. Better, you should eliminate the static variables entirely.

Here's an example:

```
// This class has a race condition in it.
public class PinballSimulator {

    private static PinballSimulator simulator = null;
    // invariant: there should never be more than one PinballSimulator
    //             object created

    private PinballSimulator() {
        System.out.println("created a PinballSimulator object");
    }

    // factory method that returns the sole PinballSimulator object,
    // creating it if it doesn't exist
    public static PinballSimulator getInstance() {
        if (simulator == null) {
            simulator = new PinballSimulator();
        }
        return simulator;
    }
}
```

This class has a race in the `getInstance()` method – two threads could call it at the same time and end up creating two copies of the `PinballSimulator` object, which we don't want.

To fix this race using the thread confinement approach, you would specify that only a certain thread (maybe the “pinball simulation thread”) is allowed to call `PinballSimulator.getInstance()`. The risk here is that Java won't help you guarantee this.

In general, static variables are very risky for concurrency. They might be hiding behind an innocuous function that seems to have no side-effects or mutations. Consider this example:

```
// is this method threadsafe?
/**
 * @param x integer to test for primeness; requires x > 1
 * @return true if x is prime with high probability
 */
public static boolean isPrime(int x) {
    if (cache.containsKey(x)) return cache.get(x);
    boolean answer = BigInteger.valueOf(x).isProbablePrime(100);
    cache.put(x, answer);
    return answer;
}

private static Map<Integer, Boolean> cache = new HashMap<>();
```

This function stores the answers from previous calls in case they're requested again. This technique is called memoization (<https://en.wikipedia.org/wiki/Memoization>) , and it's a sensible optimization for slow functions like exact primality testing. But now the `isPrime` method is not safe to call from multiple threads, and its clients may not even realize it. The reason is that the `HashMap` referenced by the static variable `cache` is shared by all calls to `isPrime()` , and `HashMap` is not threadsafe. If multiple

threads mutate the map at the same time, by calling `cache.put()`, then the map can become corrupted in the same way that the bank account became corrupted in the last reading (./19-concurrency/#shared_memory_example). If you're lucky, the corruption may cause an exception deep in the hash map, like a `NullPointerException` or `IndexOutOfBoundsException`. But it also may just quietly give wrong answers, as we saw in the bank account example (./19-concurrency/#shared_memory_example).

Strategy 2: Immutability

Our second way of achieving thread safety is by using immutable references and data types. Immutability tackles the shared-mutable-data cause of a race condition and solves it simply by making the shared data *not mutable*.

Final variables are immutable references, so a variable declared final is safe to access from multiple threads. You can only read the variable, not write it. Be careful, because this safety applies only to the variable itself, and we still have to argue that the object the variable points to is immutable.

Immutable objects are usually also threadsafe. We say "usually" here because our current definition of immutability is too loose for concurrent programming. We've said that a type is immutable if an object of the type always represents the same abstract value for its entire lifetime. But that actually allows the type the freedom to mutate its rep, as long as those mutations are invisible to clients. We saw an example of this notion, called benevolent or beneficent mutation, when we looked at an immutable list that cached its length in a mutable field (./16-recursive-data-types/recursive/#tuning_the_rep) the first time the length was requested by a client. Caching is a typical kind of beneficent mutation.

For concurrency, though, this kind of hidden mutation is not safe. An immutable data type that uses beneficent mutation will have to make itself threadsafe using locks (the same technique required of mutable data types), which we'll talk about in a future reading.

Stronger definition of immutability

So in order to be confident that an immutable data type is threadsafe without locks, we need a stronger definition of immutability:

- no mutator methods
- all fields are private and final
- no representation exposure (./13-abstraction-functions-rep-invariants/#invariants)
- no mutation whatsoever of mutable objects in the rep – not even beneficent mutation (./16-recursive-data-types/recursive/#tuning_the_rep)

If you follow these rules, then you can be confident that your immutable type will also be threadsafe.

In the Java Tutorials, read:

- **A Strategy for Defining Immutable Objects**
[\(1 page\)](https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html)

Strategy 3: Using Threadsafe Data Types

Our third major strategy for achieving thread safety is to store shared mutable data in existing threadsafe data types.

When a data type in the Java library is threadsafe, its documentation will explicitly state that fact. For example, here's what `StringBuffer` ([//docs.oracle.com/javase/8/docs/api/?java/lang/StringBuffer.html](https://docs.oracle.com/javase/8/docs/api/?java/lang/StringBuffer.html)) says:

[`StringBuffer` is] A thread-safe, mutable sequence of characters. A string buffer is like a `String`, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

String buffers are safe for use by multiple threads. The methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

This is in contrast to `StringBuilder` ([//docs.oracle.com/javase/8/docs/api/?java/lang/StringBuilder.html](https://docs.oracle.com/javase/8/docs/api/?java/lang/StringBuilder.html)) :

[`StringBuilder` is] A mutable sequence of characters. This class provides an API compatible with `StringBuffer`, but with no guarantee of synchronization. This class is designed for use as a drop-in replacement for `StringBuffer` in places where the string buffer was being used by a single thread (as is generally the case). Where possible, it is recommended that this class be used in preference to `StringBuffer` as it will be faster under most implementations.

It's become common in the Java API to find two mutable data types that do the same thing, one threadsafe and the other not. The reason is what this quote indicates: threadsafe data types usually incur a performance penalty compared to an unsafe type.

It's deeply unfortunate that `StringBuffer` and `StringBuilder` are named so similarly, without any indication in the name that thread safety is the crucial difference between them. It's also unfortunate that they don't share a common interface, so you can't simply swap in one implementation for the other for the times when you need thread safety. The Java collection interfaces do much better in this respect, as we'll see next.

Threadsafte Collections

The collection interfaces in Java – `List` , `Set` , `Map` – have basic implementations that are not threadsafe. The implementations of these that you've been used to using, namely `ArrayList` , `HashMap` , and `HashSet` , cannot be used safely from more than one thread.

Fortunately, just like the Collections API provides wrapper methods that make collections immutable, it provides another set of wrapper methods to make collections threadsafe, while still mutable.

These wrappers effectively make each method of the collection atomic with respect to the other methods. An **atomic action** effectively happens all at once – it doesn't interleave its internal operations with those of other actions, and none of the effects of the action are visible to other threads until the entire action is complete, so it never looks partially done.

Now we see a way to fix the `isPrime()` method we had earlier in the reading :

```
private static Map<Integer, Boolean> cache =
    Collections.synchronizedMap(new HashMap<>());
```

A few points here.

Don't circumvent the wrapper. Make sure to throw away references to the underlying non-threadsafte collection, and access it only through the synchronized wrapper. That happens automatically in the line of code above, since the new `HashMap` is passed only to `synchronizedMap()` and never stored

anywhere else. (We saw this same warning with the unmodifiable wrappers: the underlying collection is still mutable, and code with a reference to it can circumvent immutability.)

Iterators are still not threadsafe. Even though method calls on the collection itself (`get()`, `put()`, `add()`, etc.) are now threadsafe, iterators created from the collection are still not threadsafe. So you can't use `iterator()`, or the for loop syntax:

```
for (String s: lst) { ... } // not threadsafe, even if lst is a synchronized list
                           wrapper
```

The solution to this iteration problem will be to acquire the collection's lock when you need to iterate over it, which we'll talk about in a future reading.

Finally, **atomic operations aren't enough to prevent races**: the way that you use the synchronized collection can still have a race condition. Consider this code, which checks whether a list has at least one element and then gets that element:

```
if ( ! lst.isEmpty() ) { String s = lst.get(0); ... }
```

Even if you make `lst` into a synchronized list, this code still may have a race condition, because another thread may remove the element between the `isEmpty()` call and the `get()` call.

Even the `isPrime()` method still has potential races:

```
if (cache.containsKey(x)) return cache.get(x);
boolean answer = BigInteger.valueOf(x).isProbablePrime(100);
cache.put(x, answer);
```

The synchronized map ensures that `containsKey()`, `get()`, and `put()` are now atomic, so using them from multiple threads won't damage the rep invariant of the map. But those three operations can now interleave in arbitrary ways with each other, which might break the invariant that `isPrime` needs from the cache: if the cache maps an integer `x` to a value `f`, then `x` is prime if and only if `f` is true. If the cache ever fails this invariant, then we might return the wrong result.

So we have to argue that the races between `containsKey()`, `get()`, and `put()` don't threaten this invariant.

1. The race between `containsKey()` and `get()` is not harmful because we never remove items from the cache – once it contains a result for `x`, it will continue to do so.
2. There's a race between `containsKey()` and `put()`. As a result, it may end up that two threads will both test the primeness of the same `x` at the same time, and both will race to call `put()` with the answer. But both of them should call `put()` with the same answer, so it doesn't matter which one wins the race – the result will be the same.

The need to make these kinds of careful arguments about safety – even when you're using threadsafe data types – is the main reason that concurrency is hard.

In the Java Tutorials, read:

- **Wrapper Collections**
[\(1 page\)](https://docs.oracle.com/javase/tutorial/collections/implementations/wrapper.html)
- **Concurrent Collections**
[\(1 page\)](https://docs.oracle.com/javase/tutorial/essential/concurrency/collections.html)

How to Make a Safety Argument

We've seen that concurrency is hard to test and debug. So if you want to convince yourself and others that your concurrent program is correct, the best approach is to make an explicit argument that it's free from races, and write it down.

A safety argument needs to catalog all the threads that exist in your module or program, and the data that they use, and argue which of the four techniques you are using to protect against races for each data object or variable: confinement, immutability, threadsafe data types, or synchronization. When you use the last two, you also need to argue that all accesses to the data are appropriately atomic – that is, that the invariants you depend on are not threatened by interleaving. We gave one of those arguments for `isPrime` above.

Thread Safety Arguments for Data Types

Let's see some examples of how to make thread safety arguments for a data type. Remember our four approaches to thread safety: confinement, immutability, threadsafe data types, and synchronization. Since we haven't talked about synchronization in this reading, we'll just focus on the first three approaches.

Confinement is not usually an option when we're making an argument just about a data type, because you have to know what threads exist in the system and what objects they've been given access to. If the data type creates its own set of threads, then you can talk about confinement with respect to those threads. Otherwise, the threads are coming in from the outside, carrying client calls, and the data type may have no guarantees about which threads have references to what. So confinement isn't a useful argument in that case. Usually we use confinement at a higher level, talking about the system as a whole and arguing why we don't need thread safety for some of our modules or data types, because they won't be shared across threads by design.

Immutability is often a useful argument:

```
/** MyString is an immutable data type representing a string of characters. */
public class MyString {
    private final char[] a;
    // Thread safety argument:
    //   This class is threadsafe because it's immutable:
    //     - a is final
    //     - a points to a mutable char array, but that array is encapsulated
    //       in this object, not shared with any other object or exposed to a
    //       client
```

Here's another rep for `MyString` that requires a little more care in the argument:

```
/** MyString is an immutable data type representing a string of characters. */
public class MyString {
    private final char[] a;
    private final int start;
    private final int len;
    // Rep invariant:
    //   0 <= start <= a.length
    //   0 <= len <= a.length-start
    // Abstraction function:
    //   represents the string of characters a[start],...,a[start+length-1]
    // Thread safety argument:
    //   This class is threadsafe because it's immutable:
    //   - a, start, and len are final
    //   - a points to a mutable char array, which may be shared with other
    //     MyString objects, but they never mutate it
    //   - the array is never exposed to a client
```

Note that since this `MyString` rep was designed for sharing the array between multiple `MyString` objects, we have to ensure that the sharing doesn't threaten its thread safety. As long as it doesn't threaten the `MyString`'s immutability, however, we can be confident that it won't threaten the thread safety.

We also have to avoid rep exposure. Rep exposure is bad for any data type, since it threatens the data type's rep invariant. It's also fatal to thread safety.

Bad Safety Arguments

Here are some *incorrect* arguments for thread safety:

```
/** MyStringBuffer is a threadsafe mutable string of characters. */
public class MyStringBuffer {
    private String text;
    // Rep invariant:
    //   none
    // Abstraction function:
    //   represents the sequence text[0],...,text[text.length()-1]
    // Thread safety argument:
    //   text is an immutable (and hence threadsafe) String,
    //   so this object is also threadsafe
```

Why doesn't this argument work? `String` is indeed immutable and threadsafe; but the rep pointing to that string, specifically the `text` variable, is not immutable. `text` is not a final variable, and in fact it *can't* be final in this data type, because we need the data type to support insertion and deletion operations. So reads and writes of the `text` variable itself are not threadsafe. This argument is false.

Here's another broken argument:

```

public class Graph {
    private final Set<Node> nodes =
        Collections.synchronizedSet(new HashSet<>());
    private final Map<Node, Set<Node>> edges =
        Collections.synchronizedMap(new HashMap<>());
    // Rep invariant:
    //   for all x, y such that y is a member of edges.get(x),
    //       x, y are both members of nodes
    // Abstraction function:
    //   represents a directed graph whose nodes are the set of nodes
    //       and whose edges are the set (x,y) such that
    //           y is a member of edges.get(x)
    // Thread safety argument:
    //   - nodes and edges are final, so those variables are immutable
    //   and threadsafe
    //   - nodes and edges point to threadsafe set and map data types

```

This is a graph data type, which stores its nodes in a set and its edges in a map. (Quick quiz: is `Graph` a mutable or immutable data type? What do the `final` keywords have to do with its mutability?) `Graph` relies on other threadsafe data types to help it implement its rep – specifically the threadsafe `Set` and `Map` wrappers that we talked about above. That prevents some race conditions, but not all, because the graph's rep invariant includes a relationship *between* the node set and the edge map. All nodes that appear in the edge map also have to appear in the node set. So there may be code like this:

```

public void addEdge(Node from, Node to) {
    if ( ! edges.containsKey(from) ) {
        edges.put(from, Collections.synchronizedSet(new HashSet<>()));
    }
    edges.get(from).add(to);
    nodes.add(from);
    nodes.add(to);
}

```

This code has a race condition in it. There is a crucial moment when the rep invariant is violated, right after the `edges` map is mutated, but just before the `nodes` set is mutated. Another operation on the graph might interleave at that moment, discover the rep invariant broken, and return wrong results. Even though the threadsafe set and map data types guarantee that their own `add()` and `put()` methods are atomic and noninterfering, they can't extend that guarantee to *interactions* between the two data structures. So the rep invariant of `Graph` is not safe from race conditions. Just using immutable and threadsafe-mutable data types is not sufficient when the rep invariant depends on relationships *between* objects in the rep.

We'll have to fix this with synchronization, and we'll see how in a future reading.

Serializability

Look again at the code for the exercise above. We might also be concerned that `clear` and `insert` could interleave such that a client sees `clear` violate its postcondition.

Suppose two threads are sharing `MyStringBuffer` `sb` representing "z". They run

A	B
call <code>sb.clear()</code>	
	call <code>sb.insert(0, "a")</code>

clear and insert concurrently as shown on the right.

Thread A's assertion will fail, but not because clear violated its postcondition.

Indeed, when all the code in clear has finished running, the postcondition is satisfied.

A	B
— in clear : text = ""	— in insert : text = "" + "a" + "z"
— clear returns	— insert returns
	assert sb.toString() .equals("")

The real problem is that thread A has not anticipated possible interleaving between clear() and the assert . With any threadsafe mutable type where atomic mutators are called concurrently, *some* mutation has to “win” by being the last one applied. The result that thread A observed is identical to the execution below, where the mutators don't interleave at all:

A	B
call sb.clear()	
— in clear : text = ""	
— clear returns	
	call sb.insert(0, "a")
	— in insert : text = "" + "a" + "z"
	— insert returns
assert sb.toString() .equals("")	

What we demand from a threadsafe data type is that when clients call its atomic operations concurrently, the results are consistent with *some* sequential ordering of the calls. In this case, clearing and inserting, that means either clear -followed-by- insert , or insert -followed-by- clear . This property is called **serializability** (<https://en.wikipedia.org/wiki/Serializability>) : for any set of operations executed concurrently, the result (the values and state observable by clients) must be a result given by *some* sequential ordering of those operations.

Summary

This reading talked about three major ways to achieve safety from race conditions on shared mutable data:

- Confinement: not sharing the data.
- Immutability: sharing, but keeping the data immutable.
- Threadsafe data types: storing the shared mutable data in a single threadsafe datatype.

These ideas connect to our three key properties of good software as follows:

- **Safe from bugs.** We're trying to eliminate a major class of concurrency bugs, race conditions, and eliminate them by design, not just by accident of timing.
- **Easy to understand.** Applying these general, simple design patterns is far more understandable than a complex argument about which thread interleavings are possible and which are not.

- **Ready for change.** We're writing down these justifications explicitly in a thread safety argument, so that maintenance programmers know what the code depends on for its thread safety.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 21: Sockets & Networking

Client/server design pattern

Network sockets

I/O

Blocking

Using network sockets

Wire protocols

Testing client/server code

Summary

Reading 21: Sockets & Networking

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

In this reading we examine *client/server communication* over the network using the *socket abstraction*.

Network communication is inherently concurrent, so building clients and servers will require us to reason about their concurrent behavior and to implement them with thread safety. We must also design the *wire protocol* that clients and servers use to communicate, just as we design the operations that clients of an ADT use to work with it.

Some of the operations with sockets are *blocking* : they block the progress of a thread until they can return a result. Blocking makes writing some code easier, but it also foreshadows a new class of concurrency bugs we'll soon contend with in depth: deadlocks.

Client/server design pattern

In this reading (and in the problem set) we explore the **client/server design pattern** for communication with message passing.

In this pattern there are two kinds of processes: clients and servers. A client initiates the communication by connecting to a server. The client sends requests to the server, and the server sends replies back. Finally, the client disconnects. A server might handle connections from many clients concurrently, and clients might also connect to multiple servers.

Many Internet applications work this way: web browsers are clients for web servers, an email program like Outlook is a client for a mail server, etc.

On the Internet, client and server processes are often running on different machines, connected only by the network, but it doesn't have to be that way — the server can be a process running on the same machine as the client.

Network sockets

IP addresses

A network interface is identified by an IP address ([//en.wikipedia.org/wiki/IP_address](https://en.wikipedia.org/wiki/IP_address)) . IPv4 addresses are 32-bit numbers written in four 8-bit parts. For example (as of this writing):

- 18.9.22.69 is the IP address of a MIT web server. Every address whose first octet is 18 ([//en.wikipedia.org/wiki/List_of_assigned_/_8_IPv4_address_blocks](https://en.wikipedia.org/wiki/List_of_assigned_/_8_IPv4_address_blocks)) is on the MIT network.
- 18.9.25.15 is the address of a MIT incoming email handler.
- 173.194.123.40 is the address of a Google web server.
- 127.0.0.1 is the loopback ([//en.wikipedia.org/wiki/Loopback](https://en.wikipedia.org/wiki/Loopback)) or localhost ([//en.wikipedia.org/wiki/Localhost](https://en.wikipedia.org/wiki/Localhost)) address: it always refers to the local machine. Technically, any address whose first octet is 127 is a loopback address, but 127.0.0.1 is standard.

You can ask Google for your current IP address (<https://www.google.com/search?q=my+ip>) . In general, as you carry around your laptop, every time you connect your machine to the network it can be assigned a new IP address.

Hostnames

Hostnames ([//en.wikipedia.org/wiki/Hostname](https://en.wikipedia.org/wiki/Hostname)) are names that can be translated into IP addresses. A single hostname can map to different IP addresses at different times; and multiple hostnames can map to the same IP address. For example:

- web.mit.edu is the name for MIT's web server. You can translate this name to an IP address yourself using dig , host , or nslookup on the command line, e.g.:

```
$ dig +short web.mit.edu
18.9.22.69
```

- dmz-mailsec-scanner-4.mit.edu is the name for one of MIT's spam filter machines responsible for handling incoming email.
- google.com is exactly what you think it is. Try using one of the commands above to find google.com 's IP address. What do you see?
- localhost is a name for 127.0.0.1 . When you want to talk to a server running on your own machine, talk to localhost .

Translation from hostnames to IP addresses is the job of the Domain Name System (DNS) ([//en.wikipedia.org/wiki/Domain_Name_System](https://en.wikipedia.org/wiki/Domain_Name_System)) . It's super cool, but not part of our discussion today.

Port numbers

A single machine might have multiple server applications that clients wish to connect to, so we need a way to direct traffic on the same network interface to different processes.

Network interfaces have multiple ports ([//en.wikipedia.org/wiki/Port_\(computer_networking\)](https://en.wikipedia.org/wiki/Port_(computer_networking))) identified by a 16-bit number from 0 (which is reserved, so we effectively start at 1) to 65535.

A server process binds to a particular port — it is now **listening** on that port. Clients have to know which port number the server is listening on. There are some well-known ports ([//en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers#Well-known_ports](https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers#Well-known_ports)) which are reserved for system-level processes and provide standard ports for certain services. For example:

- Port 22 is the standard SSH port. When you connect to `athena.dialup.mit.edu` using SSH, the software automatically uses port 22.
- Port 25 is the standard email server port.
- Port 80 is the standard web server port. When you connect to the URL `http://web.mit.edu` in your web browser, it connects to `18.9.22.69` on port 80.

When the port is not a standard port, it is specified as part of the address. For example, the URL `http://128.2.39.10:9000` refers to port 9000 on the machine at `128.2.39.10`.

When a client connects to a server, that outgoing connection also uses a port number on the client's network interface, usually chosen at random from the available *non*-well-known ports.

Network sockets

A **socket** ([//en.wikipedia.org/wiki/Network_socket](https://en.wikipedia.org/wiki/Network_socket)) represents one end of the connection between client and server.

- A **listening socket** is used by a server process to wait for connections from remote clients.
In Java, use `ServerSocket` ([//docs.oracle.com/javase/8/docs/api/?java/net/ServerSocket.html](https://docs.oracle.com/javase/8/docs/api/?java/net/ServerSocket.html)) to make a listening socket, and use its `accept` ([//docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html#accept--](https://docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html#accept--)) method to listen to it.
- A **connected socket** can send and receive messages to and from the process on the other end of the connection. It is identified by both the local IP address and port number plus the remote address and port, which allows a server to differentiate between concurrent connections from different IPs, or from the same IP on different remote ports.
In Java, clients use a `Socket` ([//docs.oracle.com/javase/8/docs/api/?java/net/Socket.html](https://docs.oracle.com/javase/8/docs/api/?java/net/Socket.html)) constructor to establish a socket connection to a server. Servers obtain a connected socket as a `Socket` object returned from `ServerSocket.accept`.

I/O

Buffers

The data that clients and servers exchange over the network is sent in chunks. These are rarely just byte-sized chunks, although they might be. The sending side (the client sending a request or the server sending a response) typically writes a large chunk (maybe a whole string like "HELLO, WORLD!" or maybe 20 megabytes of video data). The network chops that chunk up into packets, and each packet is routed separately over the network. At the other end, the receiver reassembles the packets together into a stream of bytes.

The result is a bursty kind of data transmission — the data may already be there when you want to read them, or you may have to wait for them to arrive and be reassembled.

When data arrive, they go into a **buffer**, an array in memory that holds the data until you read it.

* see **What if Dr. Seuss Did Technical Writing?** ([//web.mit.edu/adorai/www/seuss-technical-writing.html](http://web.mit.edu/adorai/www/seuss-technical-writing.html)) , although the issue described in the first stanza is no longer relevant with the obsolescence of floppy disk drives

Streams

The data going into or coming out of a socket is a **stream** ([//en.wikipedia.org/wiki/Stream_\(computing\)](https://en.wikipedia.org/wiki/Stream_(computing))) of bytes.

In Java, `InputStream` ([//docs.oracle.com/javase/8/docs/api/?java/io/InputStream.html](https://docs.oracle.com/javase/8/docs/api/?java/io/InputStream.html)) objects represent sources of data flowing into your program. For example:

- Reading from a file on disk with a `FileInputStream` ([//docs.oracle.com/javase/8/docs/api/?java/io/FileInputStream.html](https://docs.oracle.com/javase/8/docs/api/?java/io/FileInputStream.html))
- User input from `System.in` ([//docs.oracle.com/javase/8/docs/api/java/lang/System.html#in](https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#in))
- Input from a network socket

`OutputStream` ([//docs.oracle.com/javase/8/docs/api/?java/io/OutputStream.html](https://docs.oracle.com/javase/8/docs/api/?java/io/OutputStream.html)) objects represent data sinks, places we can write data to. For example:

- `FileOutputStream` ([//docs.oracle.com/javase/8/docs/api/?java/io/FileOutputStream.html](https://docs.oracle.com/javase/8/docs/api/?java/io/FileOutputStream.html)) for saving to files
- `System.out` ([//docs.oracle.com/javase/8/docs/api/java/lang/System.html#out](https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#out)) is a `PrintStream` ([//docs.oracle.com/javase/8/docs/api/?java/io/PrintStream.html](https://docs.oracle.com/javase/8/docs/api/?java/io/PrintStream.html)) , an `OutputStream` that prints readable representations of various types
- Output to a network socket

In the Java Tutorials, read:

- **I/O Streams** ([//docs.oracle.com/javase/tutorial/essential/io/streams.html](https://docs.oracle.com/javase/tutorial/essential/io/streams.html)) up to and including *I/O from the Command Line* (8 pages)

With sockets, remember that the *output* of one process is the *input* of another process. If Alice and Bob have a socket connection, Alice has an output stream that flows to Bob's input stream, and *vice versa* .

Blocking

Blocking means that a thread waits (without doing further work) until an event occurs. We can use this term to describe methods and method calls: if a method is a **blocking method** , then a call to that method can **block** , waiting until some event occurs before it returns to the caller.

Socket input/output streams exhibit blocking behavior:

- When an incoming socket's buffer is empty, calling `read` blocks until data are available.
- When the destination socket's buffer is full, calling `write` blocks until space is available.

Blocking is very convenient from a programmer's point of view, because the programmer can write code as if the `read` (or `write`) call will always work, no matter what the timing of data arrival. If data (or for `write` , space) is already available in the buffer, the call might return very quickly. But if the `read` or `write` can't succeed, the call **blocks** . The operating system takes care of the details of delaying that thread until `read` or `write` can succeed.

Blocking happens throughout concurrent programming, not just in I/O ([//en.wikipedia.org/wiki/Input/output](https://en.wikipedia.org/wiki/Input/output)) (communication into and out of a process, perhaps over a network, or to/from a file, or with the user on the command line or a GUI, ...). Concurrent modules don't work in

lockstep, like sequential programs do, so they typically have to wait for each other to catch up when coordinated action is required.

We'll see in the next reading that this waiting gives rise to the second major kind of bug (the first was race conditions) in concurrent programming: **deadlock**, where modules are waiting for each other to do something, so none of them can make any progress. But that's for next time.

Using network sockets

Make sure you've read about streams at the Java Tutorial link above, then read about network sockets:

In the Java Tutorials, read:

- **All About Sockets** ([//docs.oracle.com/javase/tutorial/networking/sockets/index.html](https://docs.oracle.com/javase/tutorial/networking/sockets/index.html)) (4 pages)

This reading describes everything you need to know about creating server- and client-side sockets and writing to and reading from their I/O streams.

On the second page

The example uses a syntax we haven't seen: the try-with-resources ([//docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html](https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html)) statement. This statement has the form:

```
try (
    // create new objects here that require cleanup after being used,
    // and assign them to variables
) {
    // code here runs with those variables
    // cleanup happens automatically after the code completes
} catch(...) {
    // you can include catch clauses if the code might throw exceptions
}
```

On the last page

Notice how both `ServerSocket.accept()` and `in.readLine()` are *blocking*. This means that the server will need a *new thread* to handle I/O with each new client. While the client-specific thread is working with that client (perhaps blocked in a read or a write), another thread (perhaps the main thread) is blocked waiting to accept a new connection.

Unfortunately, their multithreaded Knock Knock Server implementation creates that new thread by *subclassed Thread*. That's *not* the recommended strategy. Instead, create a new class that implements `Runnable`, or use an anonymous `Runnable` ([./19-concurrency/anonymous-runnable/](#)) that calls a method where that client connection will be handled until it's closed. Don't use `extends Thread`. And while subclassing was popular when the Java API was designed, we don't discuss or recommend it at all because it has many downsides.

Wire protocols

Now that we have our client and server connected up with sockets, what do they pass back and forth over those sockets?

A **protocol** is a set of messages that can be exchanged by two communicating parties. A **wire protocol** in particular is a set of messages represented as byte sequences, like `hello world` and `bye` (assuming we've agreed on a way to encode those characters into bytes).

Most Internet applications use simple ASCII-based wire protocols. You can use a program called Telnet to check them out. For example:

HTTP

Hypertext Transfer Protocol (HTTP) ([//en.wikipedia.org/wiki/Hypertext_Transfer_Protocol](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol)) is the language of the World Wide Web. We already know that port 80 is the well-known port for speaking HTTP to web servers, so let's talk to one on the command line.

You'll be using Telnet on the problem set, so try these out now. User input is shown in **green**, and for input to the telnet connection, newlines (pressing enter) are shown with `↵`:

```
$ telnet www.eecs.mit.edu 80
Trying 18.62.0.96...
Connected to eecsweb.mit.edu.
Escape character is '^]'.
GET /↵
<!DOCTYPE html>
... lots of output ...
<title>Homepage | MIT EECS</title>
... lots more output ...
```

The `GET` command gets a web page. The `/` is the path of the page you want on the site. So this command fetches the page at `http://www.eecs.mit.edu:80/`. Since 80 is the default port for HTTP, this is equivalent to visiting `http://www.eecs.mit.edu` ([//www.eecs.mit.edu](http://www.eecs.mit.edu)) in your web browser. The result is HTML code that your browser renders to display the EECS homepage.

Internet protocols are defined by RFC specifications ([//en.wikipedia.org/wiki/Request_for_Comments](https://en.wikipedia.org/wiki/Request_for_Comments)) (RFC stands for “request for comment”, and some RFCs are eventually adopted as standards). RFC 1945 ([//tools.ietf.org/html/rfc1945](https://tools.ietf.org/html/rfc1945)) defined HTTP version 1.0, and was superseded by HTTP 1.1 in RFC 2616 ([//tools.ietf.org/html/rfc2616](https://tools.ietf.org/html/rfc2616)). So for many web sites, you might need to speak HTTP 1.1 if you want to talk to them. For example:

```
$ telnet web.mit.edu 80
Trying 18.9.22.69...
Connected to web.mit.edu.
Escape character is '^]'.
GET /aboutmit/ HTTP/1.1↵
Host: web.mit.edu↵
↵
HTTP/1.1 200 OK
Date: Tue, 31 Mar 2015 15:14:22 GMT
... more headers ...

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/T
... more HTML ...
<title>MIT – About</title>
... lots more HTML ...
```

This time, your request must end with a blank line. HTTP version 1.1 requires the client to specify some extra information (called headers) with the request, and the blank line signals the end of the headers.

You will also more than likely find that telnet does not exit after making this request — this time, the server keeps the connection open so you can make another request right away. To quit Telnet manually, type the escape character (probably `Ctrl -]`) to bring up the `telnet>` prompt, and type `quit` :

```
... lots more HTML ...
</html>
Ctrl-]↵
telnet> quit↵
Connection closed.
```

SMTP

Simple Mail Transfer Protocol (SMTP) ([//en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol](https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol)) is the protocol for sending email (different protocols are used for client programs that retrieve email from your inbox). Because the email system was designed in a time before spam, modern email communication is fraught with traps and heuristics designed to prevent abuse. But we can still try to speak SMTP. Recall that the well-known SMTP port is 25, and `dmz-mailsec-scanner-4.mit.edu` was the name of a MIT email handler.

You'll need to fill in `your-IP-address-here` and `your-username-here`, and the ↵ indicate newlines for clarity. This will only work if you're on MITnet, and even then your mail might be rejected for looking suspicious:

```
$ telnet dmz-mailsec-scanner-4.mit.edu 25
Trying 18.9.25.15...
Connected to dmz-mailsec-scanner-4.mit.edu.
Escape character is '^]'.
220 dmz-mailsec-scanner-4.mit.edu ESMTP Symantec Messaging Gateway
HELO your-IP-address-here↵
250 2.0.0 dmz-mailsec-scanner-4.mit.edu says HELO to your-ip-address:port
MAIL FROM: <your-username-here@mit.edu>↵
250 2.0.0 MAIL FROM accepted
RCPT TO: <your-username-here@mit.edu>↵
250 2.0.0 RCPT TO accepted
DATA↵
354 3.0.0 continue. finished with "\r\n.\r\n"
From: <your-username-here@mit.edu>↵
To: <your-username-here@mit.edu>↵
Subject: testing↵
This is a hand-crafted artisanal email.↵
.↵
250 2.0.0 OK 99/00-11111-22222222
QUIT↵
221 2.3.0 dmz-mailsec-scanner-4.mit.edu closing connection
Connection closed by foreign host.
```

SMTP is quite chatty in comparison to HTTP, providing some human-readable instructions like `continue`. `finished with "\r\n.\r\n"` to tell us how to terminate our message content.

Designing a wire protocol

When designing a wire protocol, apply the same rules of thumb you use for designing the operations of an abstract data type:

- Keep the number of different messages **small**. It's better to have a few commands and responses that can be combined rather than many complex messages.
- Each message should have a well-defined purpose and **coherent** behavior.
- The set of messages must be **adequate** for clients to make the requests they need to make and for servers to deliver the results.

Just as we demand representation independence from our types, we should aim for **platform-independence** in our protocols. HTTP can be spoken by any web server and any web browser on any operating system. The protocol doesn't say anything about how web pages are stored on disk, how they are prepared or generated by the server, what algorithms the client will use to render them, etc.

We can also apply the three big ideas in this class:

- **Safe from bugs**

- The protocol should be easy for clients and servers to generate and parse. Simpler code for reading and writing the protocol (whether written with a parser generator like ANTLR, with regular expressions, etc.) will have fewer opportunities for bugs.
- Consider the ways a broken or malicious client or server could stuff garbage data into the protocol to break the process on the other end.

Email spam is one example: when we spoke SMTP above, the mail server asked us to say who was sending the email, and there's nothing in SMTP to prevent us from lying outright. We've had to build systems on top of SMTP to try to stop spammers who lie about `From:` addresses.

Security vulnerabilities are a more serious example. For example, protocols that allow a client to send requests with arbitrary amounts of data require careful handling on the server to avoid running out of buffer space, or worse ([//en.wikipedia.org/wiki/Buffer_overflow](https://en.wikipedia.org/wiki/Buffer_overflow)) .

- **Easy to understand** : for example, choosing a text-based protocol means that we can debug communication errors by reading the text of the client/server exchange. It even allows us to speak the protocol "by hand" as we saw above.
- **Ready for change** : for example, HTTP includes the ability to specify a version number, so clients and servers can agree with one another which version of the protocol they will use. If we need to make changes to the protocol in the future, older clients or servers can continue to work by announcing the version they will use.

Serialization ([//en.wikipedia.org/wiki/Serialization](https://en.wikipedia.org/wiki/Serialization)) is the process of transforming data structures in memory into a format that can be easily stored or transmitted (not the same as serializability from *Thread Safety* ([../20-thread-safety/#serializability](#))). Rather than invent a new format for serializing your data between clients and servers, use an existing one. For example, JSON (JavaScript Object Notation) ([//en.wikipedia.org/wiki/JSON](https://en.wikipedia.org/wiki/JSON)) is a simple, widely-used format for serializing basic values, arrays, and maps with string keys.

Specifying a wire protocol

In order to precisely define for clients & servers what messages are allowed by a protocol, use a grammar.

For example, here is a very small part of the HTTP 1.1 request grammar from RFC 2616 section 5 ([//www.w3.org/Protocols/rfc2616/rfc2616-sec5.html](https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html)) :

```

request ::= request-line
          ((general-header | request-header | entity-header) CRLF)*
          CRLF
          message-body?
request-line ::= method SPACE request-uri SPACE http-version CRLF
method ::= "OPTIONS" | "GET" | "HEAD" | "POST" | ...
...
  
```

Using the grammar, we can see that in this example request from earlier:

```

GET /aboutmit/ HTTP/1.1
Host: web.mit.edu
  
```

- GET is the method : we're asking the server to get a page for us.
- /aboutmit/ is the request-uri : the description of what we want to get.
- HTTP/1.1 is the http-version .
- Host: web.mit.edu is some kind of header — we would have to examine the rules for each of the ...-header options to discover which one.
- And we can see why we had to end the request with a blank line: since a single request can have multiple headers that end in CRLF (newline), we have another CRLF at the end to finish the request .
- We don't have any message-body — and since the server didn't wait to see if we would send one, presumably that only applies for other kinds of requests.

The grammar is not enough: it fills a similar role to method signatures when defining an ADT. We still need the specifications:

- **What are the preconditions of a message?** For example, if a particular field in a message is a string of digits, is any number valid? Or must it be the ID number of a record known to the server? Under what circumstances can a message be sent? Are certain messages only valid when sent in a certain sequence?
- **What are the postconditions?** What action will the server take based on a message? What server-side data will be mutated? What reply will the server send back to the client?

Testing client/server code

Remember that concurrency is hard to test and debug (..19-concurrency/#concurrency_is_hard_to_test_and_debug) . We can't reliably reproduce race conditions, and the network adds a source of latency that is entirely beyond our control. You need to design for concurrency and argue carefully for the correctness of your code.

Separate network code from data structures and algorithms

Most of the ADTs in your client/server program don't need to rely on networking. Make sure you specify, test, and implement them as separate components that are safe from bugs, easy to understand, and ready for change — in part because they don't involve any networking code.

If those ADTs will need to be used concurrently from multiple threads (for example, threads handling different client connections), our next reading will discuss your options. Otherwise, use the thread safety strategies of confinement, immutability, and existing threadsafe data types (..20-thread-safety/) .

Separate socket code from stream code

A function or module that needs to read from and write to a socket may only need access to the input/output streams, not to the socket itself. This design allows you to test the module by connecting it to streams that don't come from a socket.

Two useful Java classes for this are `ByteArrayInputStream` ([//docs.oracle.com/javase/8/docs/api/?java/io/ByteArrayInputStream.html](https://docs.oracle.com/javase/8/docs/api/?java/io/ByteArrayInputStream.html)) and `ByteArrayOutputStream` ([//docs.oracle.com/javase/8/docs/api/?java/io/ByteArrayOutputStream.html](https://docs.oracle.com/javase/8/docs/api/?java/io/ByteArrayOutputStream.html)) . Suppose we want to test this method:

```
void upperCaseLine(BufferedReader input, PrintWriter output) throws IOException
    requires: input and output are open
    effects: attempts to read a line from input
            and attempts to write that line, in upper case, to output
```

The method is normally used with a socket:

```
Socket sock = ...

// read a stream of characters from the socket input stream
BufferedReader in = new BufferedReader(new InputStreamReader(sock.getInputStream()));
// write characters to the socket output stream
PrintWriter out = new PrintWriter(sock.getOutputStream(), true);

upperCaseLine(in, out);
```

If the case conversion is a function we implement, it should already be specified, tested, and implemented separately. But now we can now also test the read/write behavior of `upperCaseLine` :

```
// fixed input stream of "dog" (line 1) and "cat" (line 2)
String inString = "dog\ncat\n";
ByteArrayInputStream inBytes = new ByteArrayInputStream(inString.getBytes());
ByteArrayOutputStream outBytes = new ByteArrayOutputStream();

// read a stream of characters from the fixed input string
BufferedReader in = new BufferedReader(new InputStreamReader(inBytes));
// write characters to temporary storage
PrintWriter out = new PrintWriter(outBytes, true);

upperCaseLine(in, out);

// check that it read the expected amount of input
assertEquals("expected input line 2 remaining", "cat", in.readLine());
// check that it wrote the expected output
assertEquals("expected upper case of input line 1", "DOG\n", outBytes.toString());
```

In this test, `inBytes` and `outBytes` are **test stubs** ([..03-testing/#unit_testing_and_stubs](#)) . To isolate and test just `upperCaseLine` , we replace the components it normally depends on (input/output streams from a socket) with components that satisfy the same spec but have canned behavior: an input stream with fixed input, and an output stream that stores the output in memory.

Testing strategies for more complex modules might use a **mock object** to simulate the behavior of a real client or server by producing entire canned sequences of interaction and asserting the correctness of each message received from the other component.

Summary

In the *client/server design pattern*, concurrency is inevitable: multiple clients and multiple servers are connected on the network, sending and receiving messages simultaneously, and expecting timely replies. A server that *blocks* waiting for one slow client when there are other clients waiting to connect to it or to receive replies will not make those clients happy. At the same time, a server that performs incorrect computations or returns bogus results because of concurrent modification to shared mutable data by different clients will not make anyone happy.

All the challenges of making our multi-threaded code **safe from bugs**, **easy to understand**, and **ready for change** apply when we design network clients and servers. These processes run concurrently with one another (if on different machines), and any server that wants to talk to multiple clients concurrently (or a client that wants to talk to multiple servers) must manage that multi-threaded communication.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 22: Queues and Message-Passing

Two models for concurrency

Message passing with threads

Implementing message passing with queues

Stopping

Thread safety arguments with message passing

Summary

Reading 22: Queues and Message-Passing

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

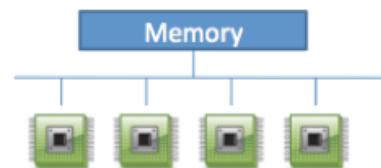
Objectives

After reading the notes and examining the code for this class, you should be able to use message passing (with synchronous queues) instead of shared memory for communication between threads.

Two models for concurrency

In our introduction to concurrency, we saw two models for concurrent programming (..19-concurrency/#two_models_for_concurrent_programming) : *shared memory* and *message passing* .

- In the **shared memory** model, concurrent modules interact by reading and writing shared mutable objects in memory. Creating multiple threads inside a single Java process is our primary example of shared-memory concurrency.
- In the **message passing** model, concurrent modules interact by sending immutable messages to one another over a communication channel. We've had one example of message passing so far: the client/server pattern (..21-sockets-networking/#clientserver_design_pattern) , in which clients and servers are concurrent processes, often on different machines, and the communication channel is a network socket (..21-sockets-networking/#network_sockets) .



The message passing model has several advantages over the shared memory model, which boil down to greater safety from bugs. In message-passing, concurrent modules interact *explicitly* , by passing messages through the communication channel, rather than *implicitly* through mutation of shared data.

The implicit interaction of shared memory can too easily lead to *inadvertent* interaction, sharing and manipulating data in parts of the program that don't know they're concurrent and aren't cooperating properly in the thread safety strategy. Message passing also shares only immutable objects (the messages) between modules, whereas shared memory *requires* sharing mutable objects, which we have already seen can be a source of bugs (./09-immutability/#risks_of_mutation).



We'll discuss in this reading how to implement message passing within a single process, as opposed to between processes over the network. We'll use **blocking queues** (an existing threadsafe type) to implement message passing between threads within a process.

Message passing with threads

We've previously talked about message passing between processes: clients and servers communicating over network sockets (./21-sockets-networking/#network_sockets). We can also use message passing between threads within the same process, and this design is often preferable to a shared memory design with locks, which we'll talk about in the next reading.

Use a synchronized queue for message passing between threads. The queue serves the same function as the buffered network communication channel in client/server message passing. Java provides the `BlockingQueue` ([//docs.oracle.com/javase/8/docs/api/?java/util/concurrent/BlockingQueue.html](https://docs.oracle.com/javase/8/docs/api/?java/util/concurrent/BlockingQueue.html)) interface for queues with blocking operations:

In an ordinary Queue ([//docs.oracle.com/javase/8/docs/api/?java/util/Queue.html](https://docs.oracle.com/javase/8/docs/api/?java/util/Queue.html)):

- `add(e)` adds element `e` to the end of the queue.
- `remove()` removes and returns the element at the head of the queue, or throws an exception if the queue is empty.

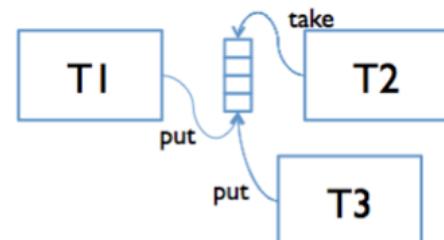
A `BlockingQueue` ([//docs.oracle.com/javase/8/docs/api/?java/util/concurrent/BlockingQueue.html](https://docs.oracle.com/javase/8/docs/api/?java/util/concurrent/BlockingQueue.html)) extends this interface:

additionaly supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

- `put(e)` blocks until it can add element `e` to the end of the queue (if the queue does not have a size bound, `put` will not block).
- `take()` blocks until it can remove and return the element at the head of the queue, waiting until the queue is non-empty.

When you are using a `BlockingQueue` for message passing between threads, make sure to use the `put()` and `take()` operations, not ~~`add()` and `remove()`~~.

Analogous to the client/server pattern for message passing over a network is the **producer-consumer design pattern** for message passing between threads. Producer threads and consumer threads share a synchronized queue. Producers put data or requests onto the queue, and consumers remove and process them. One or more producers and one or more consumers might all be adding and removing items from the same queue. This queue must be safe for concurrency.



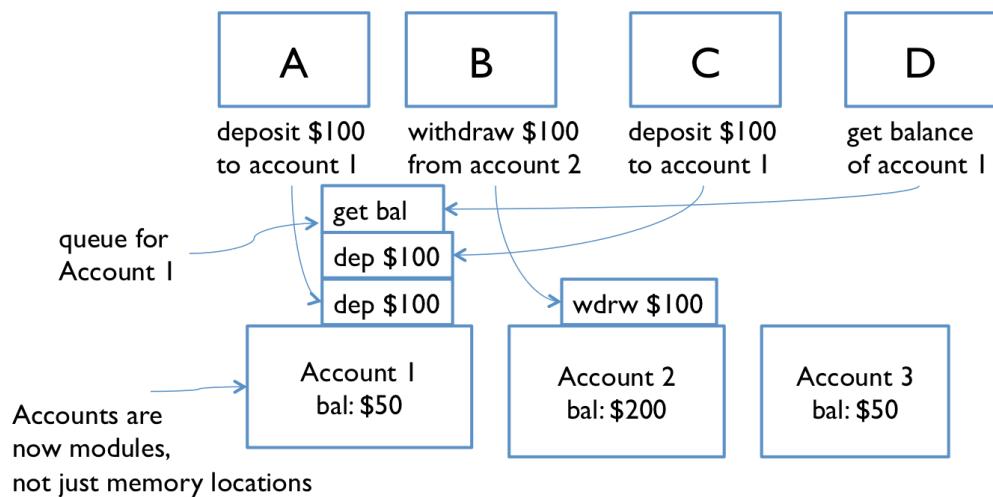
Java provides two implementations of `BlockingQueue`:

- `ArrayBlockingQueue` (<http://docs.oracle.com/javase/8/docs/api/?java/util/concurrent/ArrayBlockingQueue.html>) is a fixed-size queue that uses an array representation. Putting a new item on the queue will block if the queue is full.
- `LinkedBlockingQueue` (<http://docs.oracle.com/javase/8/docs/api/?java/util/concurrent/LinkedBlockingQueue.html>) is a growable queue using a linked-list representation. If no maximum capacity is specified, the queue will never fill up, so `put` will never block.

Unlike the streams of bytes sent and received by sockets, these synchronized queues (like normal collections classes in Java) can hold objects of an arbitrary type. Instead of designing a wire protocol, we must choose or design a type for messages in the queue. **It must be an immutable type.** And just as we did with operations on a threadsafe ADT or messages in a wire protocol, we must design our messages here to prevent race conditions and enable clients to perform the atomic operations they need.

Bank account example

Our first example of message passing was the bank account example (./19-



[concurrency/#message_passing_example](#).

Each cash machine and each account is its own module, and modules interact by sending messages to one another. Incoming messages arrive on a queue.

We designed messages for `get-balance` and `withdraw`, and said that each cash machine checks the account balance before withdrawing to prevent overdrafts:

```
get-balance
if balance >= 1 then withdraw 1
```

But it is still possible to interleave messages from two cash machines so they are both fooled into thinking they can safely withdraw the last dollar from an account with only \$1 in it.

We need to choose a better atomic operation: `withdraw-if-sufficient-funds` would be a better operation than just `withdraw`.

Implementing message passing with queues

You can see all the code for this example on GitHub: **square example** (<https://github.com/mit6005/sp16-ex22-square>). All the relevant parts are excerpted below.

Here's a message passing module for squaring integers:

`SquareQueue.java` line 6 (<https://github.com/mit6005/sp16-ex22-square/blob/master/src/square/SquareQueue.java#L6-L46>)

```

private final BlockingQueue<Integer> in;
private final BlockingQueue<SquareResult> out;
// Rep invariant: in, out != null

/** Make a new squarer.
 * @param requests queue to receive requests from
 * @param replies queue to send replies to */
public Squarer(BlockingQueue<Integer> requests,
               BlockingQueue<SquareResult> replies) {
    this.in = requests;
    this.out = replies;
}

/** Start handling squaring requests. */
public void start() {
    new Thread(new Runnable() {
        public void run() {
            while (true) {
                // TODO: we may want a way to stop the thread
                try {
                    // block until a request arrives
                    int x = in.take();
                    // compute the answer and send it back
                    int y = x * x;
                    out.put(new SquareResult(x, y));
                } catch (InterruptedException ie) {
                    ie.printStackTrace();
                }
            }
        }
    }).start();
}
}

```

Incoming messages to the `Squarer` are integers; the squarer knows that its job is to square those numbers, so no further details are required.

Outgoing messages are instances of `SquareResult` :

SquareResult.java line 48 (<https://github.com/mit6005/sp16-ex22-square/blob/master/src/square/SquareQueue.java#L48-L70>)

```

public class SquareResult {
    private final int input;
    private final int output;

    /** Make a new result message.
     * @param input input number
     * @param output square of input */
    public SquareResult(int input, int output) {
        this.input = input;
        this.output = output;
    }

    @Override public String toString() {
        return input + " ^2 = " + output;
    }
}

```

We would probably add additional observers to `SquareResult` so clients can retrieve the input number and output result.

Finally, here's a main method that uses the squarer:

SquareQueue.java line 77 (<https://github.com/mit6005/sp16-ex22-square/blob/master/src/square/SquareQueue.java#L77-L96>)

```

public static void main(String[] args) {
    BlockingQueue<Integer> requests = new LinkedBlockingQueue<>();
    BlockingQueue<SquareResult> replies = new LinkedBlockingQueue<>();

    Squarer squarer = new Squarer(requests, replies);
    squarer.start();

    try {
        // make a request
        requests.put(42);
        // ... maybe do something concurrently ...
        // read the reply
        System.out.println(replies.take());
    } catch (InterruptedException ie) {
        ie.printStackTrace();
    }
}

```

It should not surprise us that this code has a very similar flavor to the code for implementing message passing with sockets.

Stopping

What if we want to shut down the `Square` so it is no longer waiting for new inputs? In the client/server model, if we want the client or server to stop listening for our messages, we close the socket. And if we want the client or server to stop altogether, we can quit that process. But here, the square is just another thread in the same process, and we can't "close" a queue.

One strategy is a *poison pill*: a special message on the queue that signals the consumer of that message to end its work. To shut down the square, since its input messages are merely integers, we would have to choose a magic poison integer (everyone knows the square of 0 is 0 right? no one will need to ask for

the square of 0...) or use null (don't use null). Instead, we might change the type of elements on the requests queue to an ADT:

```
SquareRequest = IntegerRequest + StopRequest
```

with operations:

```
input : SquareRequest → int
shouldStop : SquareRequest → boolean
```

and when we want to stop the squarer, we enqueue a `StopRequest` where `shouldStop` returns `true`

For example, in `SquareRequest.start()` :

```
public void run() {
    while (true) {
        try {
            // block until a request arrives
            SquareRequest req = in.take();
            // see if we should stop
            if (req.shouldStop()) { break; }
            // compute the answer and send it back
            int x = req.input();
            int y = x * x;
            out.put(new SquareResult(x, y));
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
    }
}
```

It is also possible to *interrupt* a thread by calling its `interrupt()` method. If the thread is blocked waiting, the method it's blocked in will throw an `InterruptedException` (that's why we have to try-catch that exception almost any time we call a blocking method). If the thread was not blocked, an *interrupted* flag will be set. The thread must check for this flag to see whether it should stop working. For example:

```
public void run() {
    // handle requests until we are interrupted
    while ( ! Thread.interrupted() ) {
        try {
            // block until a request arrives
            int x = in.take();
            // compute the answer and send it back
            int y = x * x;
            out.put(new SquareResult(x, y));
        } catch (InterruptedException ie) {
            // stop
            break;
        }
    }
}
```

Thread safety arguments with message passing

A thread safety argument with message passing might rely on:

- **Existing threadsafe data types** for the synchronized queue. This queue is definitely shared and definitely mutable, so we must ensure it is safe for concurrency.
- **Immutability** of messages or data that might be accessible to multiple threads at the same time.
- **Confinement** of data to individual producer/consumer threads. Local variables used by one producer or consumer are not visible to other threads, which only communicate with one another using messages in the queue.
- **Confinement** of mutable messages or data that are sent over the queue but will only be accessible to one thread at a time. This argument must be carefully articulated and implemented. But if one module drops all references to some mutable data like a hot potato as soon as it puts them onto a queue to be delivered to another thread, only one thread will have access to those data at a time, precluding concurrent access.

In comparison to synchronization, message passing can make it easier for each module in a concurrent system to maintain its own thread safety invariants. We don't have to reason about multiple threads accessing shared data if the data are instead transferred between modules using a threadsafe communication channel.

Summary

- Rather than synchronize with locks, message passing systems synchronize on a shared communication channel, e.g. a stream or a queue.
- Threads communicating with blocking queues is a useful pattern for message passing within a single process.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 23: Locks and Synchronization

Introduction

Synchronization

Deadlock

Developing a threadsafe abstract data type

Locking

Monitor pattern

Thread safety argument with synchronization

Atomic operations

Designing a datatype for concurrency

Deadlock rears its ugly head

Goals of concurrent program design

Concurrency in practice

Summary

Reading 23: Locks and Synchronization

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- Understand how a lock is used to protect shared mutable data
- Be able to recognize deadlock and know strategies to prevent it
- Know the monitor pattern and be able to apply it to a data type

Introduction

Earlier, we defined thread safety (./20-thread-safety/#what_threads_mean) for a data type or a function as *behaving correctly when used from multiple threads, regardless of how those threads are executed, without additional coordination* .

Here's the general principle: **the correctness of a concurrent program should not depend on accidents of timing** .

To achieve that correctness, we enumerated four strategies for making code safe for concurrency (./20-thread-safety/) :

1. **Confinement** (./20-thread-safety/#strategy_1_confinement) : don't share data between threads.
2. **Immutability** (./20-thread-safety/#strategy_2_immutability) : make the shared data immutable.
3. **Use existing threadsafe data types** (./20-thread-safety/#strategy_3_using_threadsafe_data_types) : use a data type that does the coordination for you.
4. **Synchronization** : prevent threads from accessing the shared data at the same time. This is what we use to implement a threadsafe type, but we didn't discuss it at the time.

We talked about strategies 1-3 earlier. In this reading, we'll finish talking about strategy 4, using **synchronization** to implement your own data type that is **safe for shared-memory concurrency**.

Synchronization

The correctness of a concurrent program should not depend on accidents of timing.

Since race conditions caused by concurrent manipulation of shared mutable data are disastrous bugs — hard to discover, hard to reproduce, hard to debug — we need a way for concurrent modules that share memory to **synchronize** with each other.

Locks are one synchronization technique. A lock is an abstraction that allows at most one thread to *own* it at a time. *Holding a lock* is how one thread tells other threads: “I’m changing this thing, don’t touch it right now.”

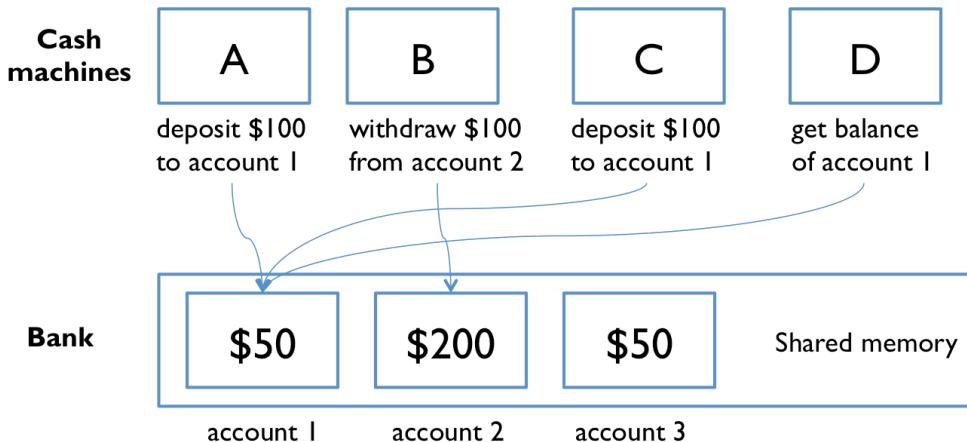
Locks have two operations:

- **acquire** allows a thread to take ownership of a lock. If a thread tries to acquire a lock currently owned by another thread, it *blocks* until the other thread releases the lock. At that point, it will contend with any other threads that are trying to acquire the lock. At most one thread can own the lock at a time.
- **release** relinquishes ownership of the lock, allowing another thread to take ownership of it.

Using a lock also tells the compiler and processor that you’re using shared memory concurrently, so that registers and caches will be flushed out to shared storage. This avoids the problem of reordering (./19-concurrency/#reordering), ensuring that the owner of a lock is always looking at up-to-date data.

Bank account example

Our first example of shared memory concurrency was a bank with cash machines (./19-



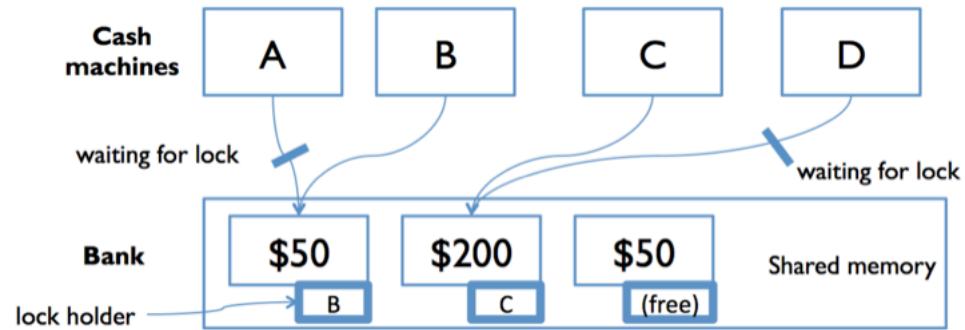
concurrency/#shared_memory_example) . The diagram from that example is on the right.

The bank has several cash machines, all of which can read and write the same account objects in memory.

Of course, without any coordination between concurrent reads and writes to the account balances, things went horribly wrong (./19-concurrency/#interleaving) .

To solve this problem with locks, we can add a lock that protects each bank account. Now, before they can access or update an account balance, cash machines must first acquire the lock on that account.

In the diagram to the right, both A and B are trying to access account 1. Suppose B acquires the lock first. Then A must wait to read and write the balance until B finishes and releases the lock. This ensures that A and B are synchronized, but another cash machine C is able to run independently on a different account (because that account is protected by a different lock).

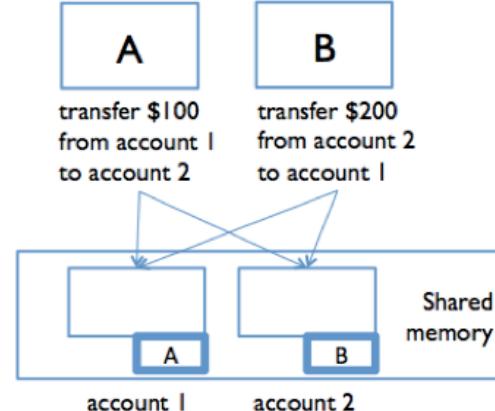


Deadlock

When used properly and carefully, locks can prevent race conditions. But then another problem rears its ugly head. Because the use of locks requires threads to wait (acquire blocks when another thread is holding the lock), it's possible to get into a situation where two threads are waiting for each other — and hence neither can make progress.

In the figure to the right, suppose A and B are making simultaneous transfers between two accounts in our bank.

A transfer between accounts needs to lock both accounts, so that money can't disappear from the system. A and B each acquire the lock on their respective "from" account: A acquires the lock on account 1, and B acquires the lock on account 2. Now, each must acquire the lock on their "to" account: so A is waiting for B to release the account 2 lock, and B is waiting for A to release the account 1 lock. Stalemate! A and B are frozen in a "deadly embrace," and accounts are locked up.



Deadlock occurs when concurrent modules are stuck waiting for each other to do something. A deadlock may involve more than two modules: the signal feature of deadlock is a **cycle of dependencies**, e.g. A is waiting for B which is waiting for C which is waiting for A. None of them can make progress.

You can also have deadlock without using any locks. For example, a message-passing system can experience deadlock when message buffers fill up. If a client fills up the server's buffer with requests, and then blocks waiting to add another request, the server may then fill up the client's buffer with results and then block itself. So the client is waiting for the server, and the server waiting for the client, and neither can make progress until the other one does. Again, deadlock ensues.

In the Java Tutorials, read:

- [Deadlock](https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html) ([//docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html](https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html)) (1 page)

Developing a threadsafe abstract data type

Let's see how to use synchronization to implement a threadsafe ADT.

You can see all the code for this example on GitHub: **edit buffer example** (<https://github.com/mit6005/sp16-ex23-editor>) . You are *not* expected to read and understand all the code. All the relevant parts are excerpted below.

Suppose we're building a multi-user editor, like Google Docs, that allows multiple people to connect to it and edit it at the same time. We'll need a mutable datatype to represent the text in the document. Here's the interface; basically it represents a string with insert and delete operations:

```
/* An EditBuffer represents a threadsafe mutable
 * string of characters in a text editor. */
editor/blob/master/src/editor/EditBuffer.java
public interface EditBuffer {

    /**
     * Modifies this by inserting a string.
     * @param pos position to insert at
     *             (requires 0 <= pos <= current buffer length)
     * @param ins string to insert
     */
    public void insert(int pos, String ins);

    /**
     * Modifies this by deleting a substring
     * @param pos starting position of substring to delete
     *             (requires 0 <= pos <= current buffer length)
     * @param len length of substring to delete
     *             (requires 0 <= len <= current buffer length - pos)
     */
    public void delete(int pos, int len);

    /**
     * @return length of text sequence in this edit buffer
     */
    public int length();

    /**
     * @return content of this edit buffer
     */
    public String toString();
}
```

A very simple rep for this datatype would just be a string:

```
SimpleBuffer.java (https://github.com/mit6005/sp16-ex23-editor/blob/master/src/editor/SimpleBuffer.java)
// Rep invariant:
//   text != null
// Abstraction function:
//   represents the sequence text[0],...,text[text.length()-1]
```

The downside of this rep is that every time we do an insert or delete, we have to copy the entire string into a new string. That gets expensive. Another rep we could use would be a character array, with space at the end. That's fine if the user is just typing new text at the end of the document (we don't have to

copy anything), but if the user is typing at the beginning of the document, then we're copying the entire document with every keystroke.

A more interesting rep, which is used by many text editors in practice, is called a *gap buffer*. It's basically a character array with extra space in it, but instead of having all the extra space at the end, the extra space is a *gap* that can appear anywhere in the buffer. Whenever an insert or delete operation needs to be done, the datatype first moves the gap to the location of the operation, and then does the insert or delete. If the gap is already there, then nothing needs to be copied — an insert just consumes part of the gap, and a delete just enlarges the gap! Gap buffers are particularly well-suited to representing a string that is being edited by a user with a cursor, since inserts and deletes tend to be focused around the cursor, so the gap rarely moves.

```
/* GapBuffer.java is a non-threadsafe EditBuffer that is optimized
 * for editing with a cursor, which tends to make a sequence of
 * inserts and deletes at the same place in the buffer. */
public class GapBuffer implements EditBuffer {
    private char[] a;
    private int gapStart;
    private int gapLength;
    // Rep invariant:
    //   a != null
    //   0 <= gapStart <= a.length
    //   0 <= gapLength <= a.length - gapStart
    // Abstraction function:
    //   represents the sequence a[0],...,a[gapStart-1],
    //   a[gapStart+gapLength],...,a[length-1]
```

In a multiuser scenario, we'd want multiple gaps, one for each user's cursor, but we'll use a single gap for now.

Steps to develop the datatype

Recall our recipe for designing and implementing an ADT:

1. **Specify.** Define the operations (method signatures and specs). We did that in the `EditBuffer` interface.
2. **Test.** Develop test cases for the operations. See `EditBufferTest` in the provided code. The test suite includes a testing strategy based on partitioning the parameter space of the operations.
3. **Rep.** Choose a rep. We chose two of them for `EditBuffer`, and this is often a good idea:
 - a. **Implement a simple, brute-force rep first.** It's easier to write, you're more likely to get it right, and it will validate your test cases and your specification so you can fix problems in them before you move on to the harder implementation. This is why we implemented `SimpleBuffer` before moving on to `GapBuffer`. Don't throw away your simple version, either — keep it around so that you have something to test and compare against in case things go wrong with the more complex one.
 - b. **Write down the rep invariant and abstraction function, and implement `checkRep()`.** `checkRep()` asserts the rep invariant at the end of every constructor, producer, and mutator method. (It's typically not necessary to call it at the end of an observer, since the rep hasn't changed.) In fact, assertions can be very useful for testing complex implementations, so it's not a bad idea to also assert the postcondition at the end of a complex method. You'll see an example of this in `GapBuffer.moveGap()` in the code with this reading.

In all these steps, we're working entirely single-threaded at first. Multithreaded clients should be in the back of our minds at all times while we're writing specs and choosing reps (we'll see later that careful choice of operations may be necessary to avoid race conditions in the clients of your datatype). But get it working, and thoroughly tested, in a sequential, single-threaded environment first.

Now we're ready for the next step:

4. **Synchronize.** Make an argument that your rep is threadsafe. Write it down explicitly as a comment in your class, right by the rep invariant, so that a maintainer knows how you designed thread safety into the class.

This part of the reading is about how to do step 4. We already saw how to make a thread safety argument ([./20-thread-safety/#how_to_make_a_safety_argument](#)) , but this time, we'll rely on synchronization in that argument.

And then the extra step we hinted at above:

5. **Iterate** . You may find that your choice of operations makes it hard to write a threadsafe type with the guarantees clients require. You might discover this in step 1, or in step 2 when you write tests, or in steps 3 or 4 when you implement. If that's the case, go back and refine the set of operations your ADT provides.

Locking

Locks are so commonly-used that Java provides them as a built-in language feature.

In Java, every object has a lock implicitly associated with it – a `String` , an array, an `ArrayList` , and every class you create, all of their object instances have a lock. Even a humble `Object` has a lock, so bare `Object` s are often used for explicit locking:

```
Object lock = new Object();
```

You can't call `acquire` and `release` on Java's intrinsic locks, however. Instead you use the **synchronized** statement to acquire the lock for the duration of a statement block:

```
synchronized (lock) { // thread blocks here until lock is free
    // now this thread has the lock
    balance = balance + 1;
    // exiting the block releases the lock
}
```

Synchronized regions like this provide **mutual exclusion** : only one thread at a time can be in a synchronized region guarded by a given object's lock. In other words, you are back in sequential programming world, with only one thread running at a time, at least with respect to other synchronized regions that refer to the same object.

Locks guard access to data

Locks are used to **guard** a shared data variable, like the account balance shown here. If all accesses to a data variable are guarded (surrounded by a synchronized block) by the same lock object, then those accesses will be guaranteed to be atomic — uninterrupted by other threads.

Because every object in Java has a lock implicitly associated with it, you might think that simply owning an object's lock would prevent other threads from accessing that object. **That is not the case.** Acquiring the lock associated with object `obj` using

```
synchronized (obj) { ... }
```

in thread t does one thing and one thing only: prevents other threads from entering a `synchronized(obj)` block, until thread t finishes its synchronized block. That's it.

Locks only provide mutual exclusion with other threads that acquire the same lock. All accesses to a data variable must be guarded by the same lock. You might guard an entire collection of variables behind a single lock, but all modules must agree on which lock they will all acquire and release.

Monitor pattern

When you are writing methods of a class, the most convenient lock is the object instance itself, i.e. `this`. As a simple approach, we can guard the entire rep of a class by wrapping all accesses to the rep inside `synchronized (this)`.

```
/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {
    private String text;
    ...
    public SimpleBuffer() {
        synchronized (this) {
            text = "";
            checkRep();
        }
    }
    public void insert(int pos, String ins) {
        synchronized (this) {
            text = text.substring(0, pos) + ins + text.substring(pos);
            checkRep();
        }
    }
    public void delete(int pos, int len) {
        synchronized (this) {
            text = text.substring(0, pos) + text.substring(pos+len);
            checkRep();
        }
    }
    public int length() {
        synchronized (this) {
            return text.length();
        }
    }
    public String toString() {
        synchronized (this) {
            return text;
        }
    }
}
```

Note the very careful discipline here. *Every* method that touches the rep must be guarded with the lock — even apparently small and trivial ones like `length()` and `toString()`. This is because reads must be guarded as well as writes — if reads are left unguarded, then they may be able to see the rep in a partially-modified state.

This approach is called the **monitor pattern**. A monitor is a class whose methods are mutually exclusive, so that only one thread can be inside an instance of the class at a time.

Java provides some syntactic sugar for the monitor pattern. If you add the keyword `synchronized` to a method signature, then Java will act as if you wrote `synchronized (this)` around the method body. So the code below is an equivalent way to implement the synchronized `SimpleBuffer` :

```
/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {
    private String text;
    ...
    public SimpleBuffer() {
        text = "";
        checkRep();
    }
    public synchronized void insert(int pos, String ins) {
        text = text.substring(0, pos) + ins + text.substring(pos);
        checkRep();
    }
    public synchronized void delete(int pos, int len) {
        text = text.substring(0, pos) + text.substring(pos+len);
        checkRep();
    }
    public synchronized int length() {
        return text.length();
    }
    public synchronized String toString() {
        return text;
    }
}
```

Notice that the `SimpleBuffer` constructor doesn't have a `synchronized` keyword. Java actually forbids it, syntactically, because an object under construction is expected to be confined to a single thread until it has returned from its constructor. So synchronizing constructors should be unnecessary.

In the Java Tutorials, read:

- **Synchronized Methods**
[\(1 page\)](https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html)
- **Intrinsic Locks and Synchronization**
[\(1 page\)](https://docs.oracle.com/javase/tutorial/essential/concurrency/locksSync.html)

Thread safety argument with synchronization

Now that we're protecting `SimpleBuffer`'s rep with a lock, we can write a better thread safety argument:

```
/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {
    private String text;
    // Rep invariant:
    //   text != null
    // Abstraction function:
    //   represents the sequence text[0],...,text[text.length()-1]
    // Thread safety argument:
    //   all accesses to text happen within SimpleBuffer methods,
    //   which are all guarded by SimpleBuffer's lock
```

The same argument works for `GapBuffer`, if we use the monitor pattern to synchronize all its methods.

Note that the encapsulation of the class, the absence of rep exposure, is very important for making this argument. If `text` were public:

```
public String text;
```

then clients outside `SimpleBuffer` would be able to read and write it without knowing that they should first acquire the lock, and `SimpleBuffer` would no longer be threadsafe.

Locking discipline

A locking discipline is a strategy for ensuring that synchronized code is threadsafe. We must satisfy two conditions:

1. Every shared mutable variable must be guarded by some lock. The data may not be read or written except inside a synchronized block that acquires that lock.
2. If an invariant involves multiple shared mutable variables (which might even be in different objects), then all the variables involved must be guarded by the *same* lock. Once a thread acquires the lock, the invariant must be reestablished before releasing the lock.

The monitor pattern as used here satisfies both rules. All the shared mutable data in the rep — which the rep invariant depends on — are guarded by the same lock.

Atomic operations

Consider a find-and-replace operation on the `EditBuffer` datatype:

```
/** Modifies buf by replacing the first occurrence of s with t.
 * If s not found in buf, then has no effect.
 * @returns true if and only if a replacement was made
 */
public static boolean findReplace(EditBuffer buf, String s, String t) {
    int i = buf.toString().indexOf(s);
    if (i == -1) {
        return false;
    }
    buf.delete(i, s.length());
    buf.insert(i, t);
    return true;
}
```

This method makes three different calls to `buf` — to convert it to a string in order to search for `s`, to delete the old text, and then to insert `t` in its place. Even though each of these calls individually is atomic, the `findReplace` method as a whole is not threadsafe, because other threads might mutate the buffer while `findReplace` is working, causing it to delete the wrong region or put the replacement back in the wrong place.

To prevent this, `findReplace` needs to synchronize with all other clients of `buf`.

Giving clients access to a lock

It's sometimes useful to make your datatype's lock available to clients, so that they can use it to implement higher-level atomic operations using your datatype.

So one approach to the problem with `findReplace` is to document that clients can use the `EditBuffer`'s lock to synchronize with each other:

```
/** An EditBuffer represents a threadsafe mutable string of characters
 * in a text editor. Clients may synchronize with each other using the
 * EditBuffer object itself. */
public interface EditBuffer {
    ...
}
```

And then `findReplace` can synchronize on `buf`:

```
public static boolean findReplace(EditBuffer buf, String s, String t) {
    synchronized (buf) {
        int i = buf.toString().indexOf(s);
        if (i == -1) {
            return false;
        }
        buf.delete(i, s.length());
        buf.insert(i, t);
        return true;
    }
}
```

The effect of this is to enlarge the synchronization region that the monitor pattern already put around the individual `toString`, `delete`, and `insert` methods, into a single atomic region that ensures that all three methods are executed without interference from other threads.

Sprinkling `synchronized` everywhere?

So is thread safety simply a matter of putting the `synchronized` keyword on every method in your program? Unfortunately not.

First, you actually don't want to synchronize methods willy-nilly. Synchronization imposes a large cost on your program. Making a synchronized method call may take significantly longer, because of the need to acquire a lock (and flush caches and communicate with other processors). Java leaves many of its mutable datatypes unsynchronized by default exactly for these performance reasons. When you don't need synchronization, don't use it.

Another argument for using `synchronized` in a more deliberate way is that it minimizes the scope of access to your lock. Adding `synchronized` to every method means that your lock is the object itself, and every client with a reference to your object automatically has a reference to your lock, that it can

acquire and release at will. Your thread safety mechanism is therefore public and can be interfered with by clients. Contrast that with using a lock that is an object internal to your rep, and acquired appropriately and sparingly using `synchronized()` blocks.

Finally, it's not actually sufficient to sprinkle `synchronized` everywhere. Dropping `synchronized` onto a method without thinking means that you're acquiring a lock without thinking about which lock it is, or about whether it's the right lock for guarding the shared data access you're about to do. Suppose we had tried to solve `findReplace`'s synchronization problem simply by dropping `synchronized` onto its declaration:

```
public static synchronized boolean findReplace(EditBuffer buf, ...) {
```

This wouldn't do what we want. It would indeed acquire a lock — because `findReplace` is a static method, it would acquire a static lock for the whole class that `findReplace` happens to be in, rather than an instance object lock. As a result, only one thread could call `findReplace` at a time — even if other threads want to operate on *different* buffers, which should be safe, they'd still be blocked until the single lock was free. So we'd suffer a significant loss in performance, because only one user of our massive multiuser editor would be allowed to do a find-and-replace at a time, even if they're all editing different documents.

Worse, however, it wouldn't provide useful protection, because other code that touches the document probably wouldn't be acquiring the same lock. It wouldn't actually eliminate our race conditions.

The `synchronized` keyword is not a panacea. Thread safety requires a discipline — using confinement, immutability, or locks to protect shared data. And that discipline needs to be written down, or maintainers won't know what it is.

Designing a datatype for concurrency

`findReplace`'s problem can be interpreted another way: that the `EditBuffer` interface really isn't that friendly to multiple simultaneous clients. It relies on integer indexes to specify insert and delete locations, which are extremely brittle to other mutations. If somebody else inserts or deletes before the index position, then the index becomes invalid.

So if we're designing a datatype specifically for use in a concurrent system, we need to think about providing operations that have better-defined semantics when they are interleaved. For example, it might be better to pair `EditBuffer` with a `Position` datatype representing a cursor position in the buffer, or even a `Selection` datatype representing a selected range. Once obtained, a `Position` could hold its location in the text against the wash of insertions and deletions around it, until the client was ready to use that `Position`. If some other thread deleted all the text around the `Position`, then the `Position` would be able to inform a subsequent client about what had happened (perhaps with an exception), and allow the client to decide what to do. These kinds of considerations come into play when designing a datatype for concurrency.

As another example, consider the `ConcurrentMap` ([//docs.oracle.com/javase/8/docs/api/?java/util/concurrent/ConcurrentMap.html](https://docs.oracle.com/javase/8/docs/api/?java/util/concurrent/ConcurrentMap.html)) interface in Java. This interface extends the existing `Map` interface, adding a few key methods that are commonly needed as atomic operations on a shared mutable map, e.g.:

- `map.putIfAbsent(key, value)` is an atomic version of
`if (! map.containsKey(key)) map.put(key, value);`
- `map.replace(key, value)` is an atomic version of
`if (map.containsKey(key)) map.put(key, value);`

Deadlock rears its ugly head

The locking approach to thread safety is powerful, but (unlike confinement and immutability) it introduces blocking into the program. Threads must sometimes wait for other threads to get out of synchronized regions before they can proceed. And blocking raises the possibility of deadlock — a very real risk, and frankly *far* more common in this setting than in message passing with blocking I/O (where we first mentioned it).

With locking, deadlock happens when threads acquire multiple locks at the same time, and two threads end up blocked while holding locks that they are each waiting for the other to release. The monitor pattern unfortunately makes this fairly easy to do. Here's an example.

Suppose we're modeling the social network of a series of books:

```
public class Wizard {
    private final String name;
    private final Set<Wizard> friends;
    // Rep invariant:
    //   name, friends != null
    //   friend links are bidirectional:
    //     for all f in friends, f.friends contains this
    // Concurrency argument:
    //   threadsafe by monitor pattern: all accesses to rep
    //   are guarded by this object's lock

    public Wizard(String name) {
        this.name = name;
        this.friends = new HashSet<Wizard>();
    }

    public synchronized boolean isFriendsWith(Wizard that) {
        return this.friends.contains(that);
    }

    public synchronized void friend(Wizard that) {
        if (friends.add(that)) {
            that.friend(this);
        }
    }

    public synchronized void defriend(Wizard that) {
        if (friends.remove(that)) {
            that.defriend(this);
        }
    }
}
```

Like Facebook, this social network is bidirectional: if x is friends with y , then y is friends with x . The `friend()` and `defriend()` methods enforce that invariant by modifying the reps of both objects, which because they use the monitor pattern means acquiring the locks to both objects as well.

Let's create a couple of wizards:

```
Wizard harry = new Wizard("Harry Potter");
Wizard snape = new Wizard("Severus Snape");
```

And then think about what happens when two independent threads are repeatedly running:

```
// thread A           // thread B
harry.friend(snape); snape.friend(harry);
harry.defriend(snape); snape.defriend(harry);
```

We will deadlock very rapidly. Here's why. Suppose thread A is about to execute `harry.friend(snape)`, and thread B is about to execute `snape.friend(harry)`.

- Thread A acquires the lock on `harry` (because the `friend` method is synchronized).
- Then thread B acquires the lock on `snape` (for the same reason).
- They both update their individual reps independently, and then try to call `friend()` on the other object — which requires them to acquire the lock on the other object.

So A is holding Harry and waiting for Snape, and B is holding Snape and waiting for Harry. Both threads are stuck in `friend()`, so neither one will ever manage to exit the synchronized region and release the lock to the other. This is a classic deadly embrace. The program simply stops.

The essence of the problem is acquiring multiple locks, and holding some of the locks while waiting for another lock to become free.

Notice that it is possible for thread A and thread B to interleave such that deadlock does not occur: perhaps thread A acquires and releases both locks before thread B has enough time to acquire the first one. If the locks involved in a deadlock are also involved in a race condition — and very often they are — then the deadlock will be just as difficult to reproduce or debug.

Deadlock solution 1: lock ordering

One way to prevent deadlock is to put an ordering on the locks that need to be acquired simultaneously, and ensuring that all code acquires the locks in that order.

In our social network example, we might always acquire the locks on the `Wizard` objects in alphabetical order by the wizard's name. Since thread A and thread B are both going to need the locks for Harry and Snape, they would both acquire them in that order: Harry's lock first, then Snape's. If thread A gets Harry's lock before B does, it will also get Snape's lock before B does, because B can't proceed until A releases Harry's lock again. The ordering on the locks forces an ordering on the threads acquiring them, so there's no way to produce a cycle in the waiting-for graph.

Here's what the code might look like:

```
public void friend(Wizard that) {
    Wizard first, second;
    if (this.name.compareTo(that.name) < 0) {
        first = this; second = that;
    } else {
        first = that; second = this;
    }
    synchronized (first) {
        synchronized (second) {
            if (friends.add(that)) {
                that.friend(this);
            }
        }
    }
}
```

(Note that the decision to order the locks alphabetically by the person's name would work fine for this book, but it wouldn't work in a real life social network. Why not? What would be better to use for lock ordering than the name?)

Although lock ordering is useful (particularly in code like operating system kernels), it has a number of drawbacks in practice.

- First, it's not modular — the code has to know about all the locks in the system, or at least in its subsystem.
- Second, it may be difficult or impossible for the code to know exactly which of those locks it will need before it even acquires the first one. It may need to do some computation to figure it out. Think about doing a depth-first search on the social network graph, for example — how would you know which nodes need to be locked, before you've even started looking for them?

Deadlock solution 2: coarse-grained locking

A more common approach than lock ordering, particularly for application programming (as opposed to operating system or device driver programming), is to use coarser locking — use a single lock to guard many object instances, or even a whole subsystem of a program.

For example, we might have a single lock for an entire social network, and have all the operations on any of its constituent parts synchronize on that lock. In the code below, all `Wizard`s belong to a `Castle`, and we just use that `Castle` object's lock to synchronize:

```
public class Wizard {
    private final Castle castle;
    private final String name;
    private final Set<Wizard> friends;
    ...
    public void friend(Wizard that) {
        synchronized (castle) {
            if (this.friends.add(that)) {
                that.friend(this);
            }
        }
    }
}
```

Coarse-grained locks can have a significant performance penalty. If you guard a large pile of mutable data with a single lock, then you're giving up the ability to access any of that data concurrently. In the worst case, having a single lock protecting everything, your program might be essentially sequential — only one thread is allowed to make progress at a time.

Goals of concurrent program design

Now is a good time to pop up a level and look at what we're doing. Recall that our primary goals are to create software that is **safe from bugs**, **easy to understand**, and **ready for change**.

Building concurrent software is clearly a challenge for all three of these goals. We can break the issues into two general classes. When we ask whether a concurrent program is *safe from bugs*, we care about two properties:

- **Safety.** Does the concurrent program satisfy its invariants and its specifications? Races in accessing mutable data threaten safety. Safety asks the question: can you prove that **some bad thing never happens**?

- **Liveness.** Does the program keep running and eventually do what you want, or does it get stuck somewhere waiting forever for events that will never happen? Can you prove that **some good thing eventually happens** ?

Deadlocks threaten liveness. Liveness may also require *fairness* , which means that concurrent modules are given processing capacity to make progress on their computations. Fairness is mostly a matter for the operating system's thread scheduler, but you can influence it (for good or for ill) by setting thread priorities.

Concurrency in practice

What strategies are typically followed in real programs?

- **Library data structures** either use no synchronization (to offer high performance to single-threaded clients, while leaving it to multithreaded clients to add locking on top) or the monitor pattern.
- **Mutable data structures with many parts** typically use either coarse-grained locking or thread confinement. Most graphical user interface toolkits follow one of these approaches, because a graphical user interface is basically a big mutable tree of mutable objects. Java Swing, the graphical user interface toolkit, uses thread confinement. Only a single dedicated thread is allowed to access Swing's tree. Other threads have to pass messages to that dedicated thread in order to access the tree.
- **Search** often uses immutable datatypes. Our Boolean formula satisfiability search (../16-recursive-data-types/recursive/#another_example_boolean_formulas) would be easy to make multithreaded, because all the datatypes involved were immutable. There would be no risk of either races or deadlocks.
- **Operating systems** often use fine-grained locks in order to get high performance, and use lock ordering to deal with deadlock problems.

We've omitted one important approach to mutable shared data because it's outside the scope of this course, but it's worth mentioning: **a database** . Database systems are widely used for distributed client/server systems like web applications. Databases avoid race conditions using *transactions* , which are similar to synchronized regions in that their effects are atomic, but they don't have to acquire locks, though a transaction may fail and be rolled back if it turns out that a race occurred. Databases can also manage locks, and handle locking order automatically. For more about how to use databases in system design, 6.170 Software Studio is strongly recommended; for more about how databases work on the inside, take 6.814 Database Systems.

And if you're interested in the **performance** of concurrent programs — since performance is often one of the reasons we add concurrency to a system in the first place — then 6.172 Performance Engineering is the course for you.

Summary

Producing a concurrent program that is safe from bugs, easy to understand, and ready for change requires careful thinking. Heisenbugs will skitter away as soon as you try to pin them down, so debugging simply isn't an effective way to achieve correct threadsafe code. And threads can interleave their operations in so many different ways that you will never be able to test even a small fraction of all possible executions.

- Make thread safety arguments about your datatypes, and document them in the code.

- Acquiring a lock allows a thread to have exclusive access to the data guarded by that lock, forcing other threads to block — as long as those threads are also trying to acquire that same lock.
- The *monitor pattern* guards the rep of a datatype with a single lock that is acquired by every method.
- Blocking caused by acquiring multiple locks creates the possibility of deadlock.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 24: Graphical User Interfaces

[View Tree](#)

[How the View Tree is Used](#)

[Input Handling](#)

[Separating Frontend from Backend](#)

[Background Processing in Graphical User Interfaces](#)

[Summary](#)

Reading 24: Graphical User Interfaces

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

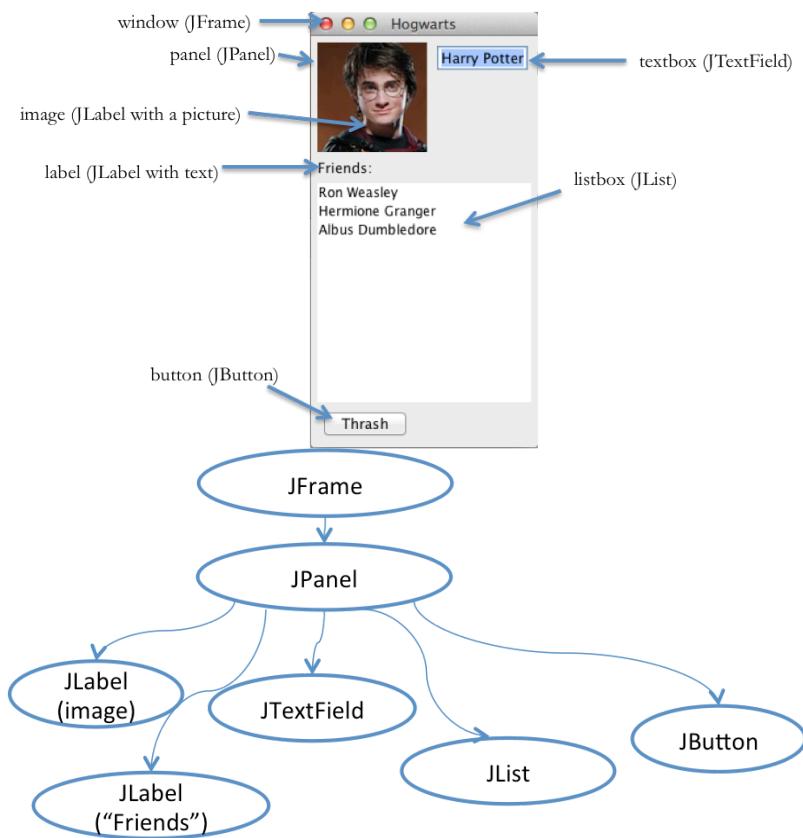
Today we'll take a high-level look at the software architecture of GUI software, focusing on the design patterns that have proven most useful. Three of the most important patterns are:

- the view tree, which is a central feature in the architecture of every important GUI toolkit;
- the model-view-controller pattern, which separates input, output, and data;
- the listener pattern, which is essential to decoupling the model from the view and controller.

View Tree

Graphical user interfaces are composed of view objects, each of which occupies a certain portion of the screen, generally a rectangular area called its bounding box. The view concept goes by a variety of names in various UI toolkits. In Java Swing ([https://en.wikipedia.org/wiki/Swing_\(Java\)](https://en.wikipedia.org/wiki/Swing_(Java))), they're `JComponent` objects; in HTML, they're elements or nodes; in other toolkits, they may be called widgets, controls, or interactors.

This leads to the first important pattern we'll talk about today: the view tree. Views are arranged into a hierarchy of containment, in which some views contain other views. Typical containers are windows, panels, and toolbars. The view tree is not just an arbitrary hierarchy, but is in fact a spatial one: child views are nested inside their parent's bounding box.



How the View Tree is Used

Virtually every GUI system has some kind of view tree. The view tree is a powerful structuring idea, which is loaded with responsibilities in a typical GUI:

Output. Views are responsible for displaying themselves, and the view tree directs the display process. GUIs change their output by mutating the view tree. For example, to show a new set of photos in a photo album GUI, the current thumbnails are removed from the view tree and a new set of thumbnails is added in their place. A redraw algorithm built into the GUI toolkit automatically redraws the affected parts of the subtree. In Java Swing, every view in the tree has a `paint()` method that knows how to draw itself on the screen. The repaint process is driven by calling `paint()` on the root of the tree, which recursively calls `paint()` down through all the descendent nodes of the view tree.

Input. Views can have input handlers, and the view tree controls how mouse and keyboard input is processed. More on this in a moment.

Layout. The view tree controls how the views are laid out on the screen, i.e. how their bounding boxes are assigned. An automatic layout algorithm automatically calculates positions and sizes of views. Specialized containers (like `JSplitPane`, `JScrollPane`) do layout themselves. More generic containers (`JPanel`, `JFrame`) delegate layout decisions to a layout manager (e.g. `GridLayout`, `BorderLayout`, `BoxLayout`, ...).

Input Handling

Input is handled somewhat differently in GUIs than we've been handling it in parsers and servers. In those systems, we've seen a single parser that peels apart the input and decides how to direct it to different modules of the program. If a GUI were written that way, it might look like this (in pseudocode):

```

while (true) {
    read mouse click
    if (clicked on Thrash button) doThrash();
    else if (clicked on textbox) doPlaceCursor();
    else if (clicked on a name in the listbox) doSelectItem();
    ...
}

```

In a GUI, we don't directly write this kind of method, because it's not modular – it mixes up responsibilities for button, listbox, and textbox all in one place. Instead, GUIs exploit the spatial separation provided by the view tree to provide functional separation as well. Mouse clicks and keyboard events are distributed around the view tree, depending on where they occur.

GUI input event handling is an instance of the **Listener pattern** (also known as Publish-Subscribe). In the Listener pattern:

- An event source generates a stream of discrete events, which correspond to state transitions in the source.
- One or more listeners register interest (subscribe) to the stream of events, providing a function to be called when a new event occurs.

In this case, the mouse is the event source, and the events are changes in the state of the mouse: its x,y position or the state of its buttons (whether they are pressed or released). Events often include additional information about the transition (such as the x,y position of mouse), which might be bundled into an event object or passed as parameters.

When an event occurs, the event source distributes it to all subscribed listeners, by calling their callback methods.

The control flow through a graphical user interface proceeds like this:

- A top-level event loop reads input from mouse and keyboard. In Java Swing, and most graphical user interface toolkits, this loop is actually hidden from you. It's buried inside the toolkit, and listeners appear to be called magically.
- For each input event, it finds the right view in the tree (by looking at the x,y position of the mouse) and sends the event to that view's listeners.
- Each listener does its thing (which might involve e.g. modifying objects in the view tree), and then *returns immediately to the event loop*.

The last part – listeners return to the event loop as fast as possible – is very important, because it preserves the responsiveness of the user interface. We'll come back to this later in the reading.

The Listener pattern isn't just used for low-level input events like mouse clicks and keyboard keypresses. Many GUI objects generate their own higher-level events, often as a result of some combination of low-level input events. For example:

- JButton sends an action event when it is pressed (whether by mouse or keyboard)
- JList sends a selection event when the selected element changes (whether by mouse or by keyboard)
- JTextField sends change events when the text inside it changes for any reason

A button can be pressed either by the mouse (with a mouse down and mouse up event) or by the keyboard (which is important for people who can't use a mouse, like blind users). So you should always listen for these high-level events, not the low-level input events. Use an ActionListener to respond to a JButton press, not a mouse listener.

Separating Frontend from Backend

We've seen how GUI programs are structured around a view tree, and how input events are handled by attaching listeners to views. This is the start of a separation of concerns – output handled by views, and input handled by listeners.

But we're still missing the application itself – the backend that represents the data and logic that the user interface is showing and editing. (Why do we want to separate this from the user interface?)

The **Model-View-Controller pattern** has this separation of concerns as its primary goal. It separates the user interface frontend from the application backend, by putting backend code into the model and frontend code into the view and controller. MVC also separates input from output; the controller is supposed to handle input, and the view is supposed to handle output.

The **model** is responsible for maintaining application-specific data and providing access to that data. Models are often mutable, and they provide methods for changing the state safely, preserving its representation invariants. OK, all mutable objects do that. But a model must also notify its clients when there are changes to its data, so that dependent views can update their displays, and dependent controllers can respond appropriately.

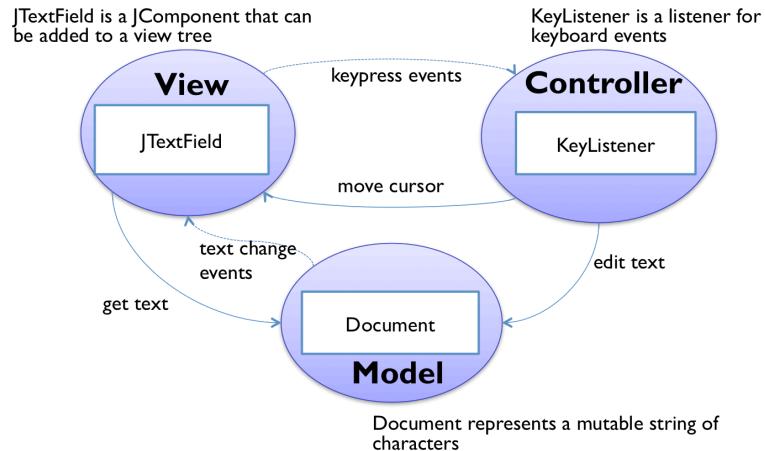
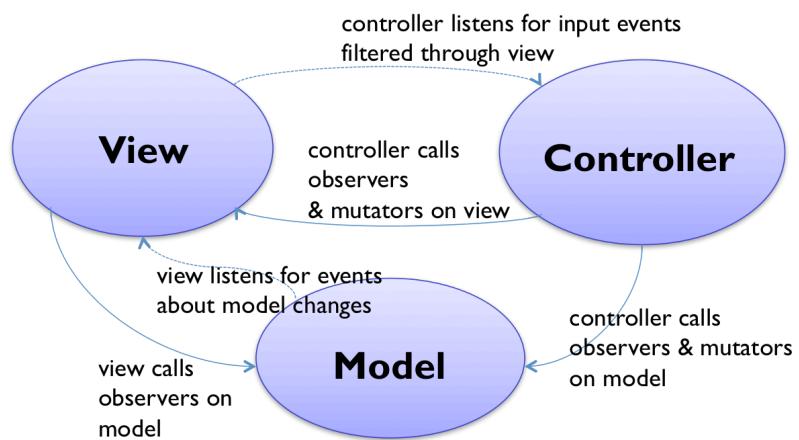
Models do this notification using the listener pattern, in which interested views and controllers register themselves as listeners for change events generated by the model.

View objects are responsible for output. A view usually occupies some chunk of the screen, usually a rectangular area. Basically, the view queries the model for data and draws the data on the screen. It listens for changes from the model so that it can update the screen to reflect those changes.

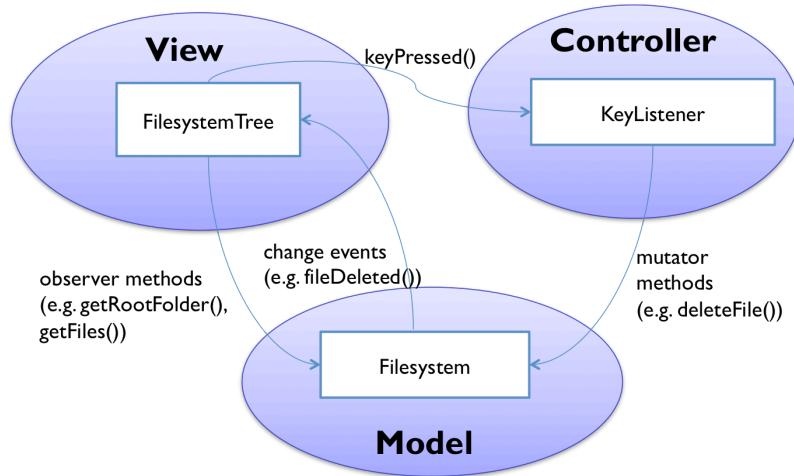
Finally, the **controller** handles the input. It receives keyboard and mouse events, and instructs the model to change accordingly.

A simple example of the MVC pattern is a text field. The figure at right shows Java Swing's text field, called `JTextField`. Its model is a mutable string of characters. The view is an object that draws the text on the screen (usually with a rectangle around it to indicate that it's an editable text field). The controller is an object that receives keystrokes typed by the user and inserts them into the mutable string.

Instances of the MVC pattern appear at many scales in GUI software. At a higher level, this text field might be part of a view (like an address book editor), with a different controller listening to it (for text-changed events), for a different model (like the address book). But when you drill down to a lower level, the text field itself is an instance of MVC.



Here's a larger example, in which the view is a filesystem browser (like the Mac Finder or Windows Explorer), the model is the disk filesystem, and the controller is an input handler that translates the user's keystrokes and mouse clicks into operations on the model and view.



The separation of model and view has several benefits. First, it allows the interface to have multiple views showing the same application data. For example, a database field might be shown in a table and in an editable form at the same time. Second, it allows views and models to be reused in other applications. The MVC pattern enables the creation of user interface toolkits, which are libraries of reusable views. Java Swing is such a toolkit. You can easily reuse view classes from this library (like JButton and JTree) while plugging your own models into them.

Background Processing in Graphical User Interfaces

The last major topic for today connects back to concurrency.

First, some motivation. Why do we need to do background processing in graphical user interfaces? Even though computer systems are steadily getting faster, we're also asking them to do more. Many programs need to do operations that may take some time: retrieving URLs over the network, running database queries, scanning a filesystem, doing complex calculations, etc.

But graphical user interfaces are event-driven programs, which means (generally speaking) everything is triggered by an input event handler. For example, in a web browser, clicking a hyperlink starts loading a new web page. But if the click handler is written so that it actually retrieves the web page itself, then the web browser will be very painful to use. Why? Because its interface will appear to freeze up until the click handler finishes retrieving the web page and returns to the event loop. Here's why.

This happens because input handling and screen repainting is all handled from a single thread. That thread (called the event-dispatch thread) has a loop that reads an input event from the queue and dispatches it to listeners on the view tree. When there are no input events left to process, it repaints the screen. But if an input handler you've written delays returning to this loop – because it's blocking on a network read, or because it's searching for the solution to a big Sudoku puzzle – then input events stop being handled, and the screen stops updating. So long tasks need to run in the background.

In Java, the event-dispatch thread is distinct from the main thread of the program (see below). It is started automatically when a user interface object is created. As a result, every Java GUI program is automatically multithreaded. Many programmers don't notice, because the main thread typically doesn't do much in a GUI program – it starts creation of the view, and then the main thread just exits, leaving only the event-dispatch thread to do the main work of the program.

The fact that Swing programs are multithreaded by default creates risks. There's very often a shared mutable datatype in your GUI: the model. If you use background threads to modify the model without blocking the event-dispatch thread, then you have to make sure your data structure is threadsafe.

But another important shared mutable datatype in your GUI is the view tree. Java Swing's view tree is not threadsafe. In general, you cannot safely call methods on a Swing object from anywhere but the event-dispatch thread.

The view tree is a big meatball of shared state, and the Swing specification doesn't guarantee that there's any lock protecting it. Instead the view tree is **confined to the event-dispatch thread**, by specification. So it's ok to access view objects from the event-dispatch thread (i.e., in response to input events), but the Swing specification forbids touching – reading or writing – any `JComponent` objects from a different thread. See Swing threading and the event-dispatch thread (<http://www.javaworld.com/javaworld/jw-08-2007/jw-08-swingthreading.html>) .

In the actual Swing implementation, there is one big lock (`Component.getTreeLock()` (<https://docs.oracle.com/javase/8/docs/api/java.awt/Component.html#getTreeLock-->)) but only some Swing methods use it, so it's not effective as a synchronization mechanism.

The safe way to access the view tree is to do it from the event-dispatch thread. So Swing takes a clever approach: it uses the event queue itself as a message-passing queue. In other words, you can put your own custom messages on the event queue, the same queue used for mouse clicks, keypresses, button action events, and so forth. Your custom message is actually a piece of executable code, an object that implements `Runnable` , and you put it on the queue using `SwingUtilities.invokeLater` (<https://docs.oracle.com/javase/8/docs/api/javax/swing/SwingUtilities.html#invokeLater-java.lang.Runnable->) . For example:

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        content.add(thumbnail);
        ...
    }
});
```

The `invokeLater()` drops this `Runnable` object at the end of the queue, and when Swing's event loop reaches it, it simply calls `run()` . Thus the body of `run()` ends up run by the event-dispatch thread, where it can safely call observers and mutators on the view tree.

In the Java Tutorials, read:

- **Concurrency in Swing**
(<https://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html>) (1 page)
- **Initial Threads** (<https://docs.oracle.com/javase/tutorial/uiswing/concurrency/initial.html>) (1 page)
- **The Event Dispatch Thread**
(<https://docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html>) (1 page)

Summary

- The view tree organizes the screen into a tree of nested rectangles, and it is used in dispatching input events as well as displaying output.
- The Listener pattern sends a stream of events (like mouse or keyboard events, or button action events) to registered listeners.
- The Model-View-Controller pattern separates responsibilities: model=data, view=output, controller=input.

- Long-running processing should be moved to a background thread, but the Swing view tree is confined to the event-dispatch thread. So accessing Swing objects from another thread requires using the event loop as a message-passing queue, to get back to the event-dispatch thread.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 25: Map, Filter, Reduce

Introduction: an example

Abstracting out control flow

Map

Functions as values

Filter

Reduce

Back to the intro example

Benefits of abstracting out control

First-class functions in Java

Map/filter/reduce in Java

Higher-order functions in Java

Summary

Reading 25: Map, Filter, Reduce

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

In this reading you'll learn a design pattern for implementing functions that operate on sequences of elements, and you'll see how treating functions themselves as *first-class values* that we can pass around and manipulate in our programs is an especially powerful idea.

- Map/filter/reduce
- Lambda expressions
- Functional objects
- Higher-order functions

Introduction: an example

Suppose we're given the following problem: write a method that finds the words in the Java files in your project.

Following good practice, we break it down into several simpler steps and write a method for each one:

- find all the files in the project, by scanning recursively from the project's root folder
- restrict them to files with a particular suffix, in this case `.java`
- open each file and read it in line-by-line
- break each line into words

Writing the individual methods for these substeps, we'll find ourselves writing a lot of low-level iteration code. For example, here's what the recursive traversal of the project folder might look like:

```
/** 
 * Find all the files in the filesystem subtree rooted at folder.
 * @param folder root of subtree, requires folder.isDirectory() == true
 * @return list of all ordinary files (not folders) that have folder as
 *         their ancestor
 */
public static List<File> allFilesIn(File folder) {
    List<File> files = new ArrayList<>();
    for (File f : folder.listFiles()) {
        if (f.isDirectory()) {
            files.addAll(allFilesIn(f));
        } else if (f.isFile()) {
            files.add(f);
        }
    }
    return files;
}
```

And here's what the filtering method might look like, which restricts that file list down to just the Java files (imagine calling this like `onlyFilesWithSuffix(files, ".java")`):

```
/** 
 * Filter a list of files to those that end with suffix.
 * @param files list of files (all non-null)
 * @param suffix string to test
 * @return a new list consisting of only those files whose names end with
 *         suffix
 */
public static List<File> onlyFilesWithSuffix(List<File> files, String suffix) {
    List<File> result = new ArrayList<>();
    for (File f : files) {
        if (f.getName().endsWith(suffix)) {
            result.add(f);
        }
    }
    return result;
}
```

→ **full Java code for the example** (<https://github.com/mit6005/sp16-ex25-words/blob/master/src/words/Words1.java>)

In this reading we discuss *map/filter/reduce*, a design pattern that substantially simplifies the implementation of functions that operate over sequences of elements. In this example, we'll have lots of sequences — lists of files; input streams that are sequences of lines; lines that are sequences of words; frequency tables that are sequences of (word, count) pairs. *Map/filter/reduce* will enable us to operate on those sequences with no explicit control flow — not a single `for` loop or `if` statement.

Along the way, we'll also see an important Big Idea: functions as "first-class" data values, meaning that they can be stored in variables, passed as arguments to functions, and created dynamically like other values.

Using first-class functions in Java is more verbose, uses some unfamiliar syntax, and the interaction with static typing adds some complexity. So to get started with map/filter/reduce, we'll switch back to Python.

Abstracting out control flow

We've already seen one design pattern that abstracts away from the details of iterating over a data structure: Iterator.

Iterator abstraction

Iterator gives you a sequence of elements from a data structure, without you having to worry about whether the data structure is a set or a token stream or a list or an array — the `Iterator` ([//docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html](https://docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html)) looks the same no matter what the data structure is.

For example, given a `List<File>` `files`, we can iterate using indices:

```
for (int ii = 0; ii < files.size(); ii++) {
    File f = files.get(ii);
    // ...
```

But this code depends on the `size` and `get` methods of `List`, which might be different in another data structure. Using an iterator abstracts away the details:

```
Iterator<File> iter = files.iterator();
while (iter.hasNext()) {
    File f = iter.next();
    // ...
```

Now the loop will be identical for any type that provides an `Iterator`. There is, in fact, an interface for such types: `Iterable` ([//docs.oracle.com/javase/8/docs/api/?java/lang/Iterable.html](https://docs.oracle.com/javase/8/docs/api/?java/lang/Iterable.html)). Any `Iterable` can be used with Java's enhanced for statement (<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html>) — `for (File f : files)` — and under the hood, it uses an iterator.

Map/filter/reduce abstraction

The map/filter/reduce patterns in this reading do something similar to Iterator, but at an even higher level: they treat the entire sequence of elements as a unit, so that the programmer doesn't have to name and work with the elements individually. In this paradigm, the control statements disappear: specifically, the `for` statements, the `if` statements, and the `return` statements in the code from our introductory example will be gone. We'll also be able to get rid of most of the temporary names (i.e., the local variables `files`, `f`, and `result`).

Sequences

Let's imagine an abstract datatype `Seq<E>` that represents a *sequence* of elements of type `E`.

For example, `[1, 2, 3, 4] ∈ Seq<Integer>`.

Any datatype that has an iterator can qualify as a sequence: array, list, set, etc. A string is also a sequence (of characters), although Java's strings don't offer an iterator. Python is more consistent in this respect: not only are lists iterable, but so are strings, tuples (which are immutable lists), and even input streams (which produce a sequence of lines). We'll see these examples in Python first, since the syntax is very readable and familiar to you, and then we'll see how it works in Java.

We'll have three operations for sequences: map, filter, and reduce. Let's look at each one in turn, and then look at how they work together.

Map

Map applies a unary function to each element in the sequence and returns a new sequence containing the results, in the same order:

map : ($E \rightarrow F$) \times Seq $<E>$ \rightarrow Seq $<F>$

For example, in Python:

```
>>> from math import sqrt
>>> map(sqrt, [1, 4, 9, 16])
[1.0, 2.0, 3.0, 4.0]
>>> map(str.lower, ['A', 'b', 'C'])
['a', 'b', 'c']
```

`map` is built-in, but it is also straightforward to implement in Python:

```
def map(f, seq):
    result = []
    for elt in seq:
        result.append(f(elt))
    return result
```

This operation captures a common pattern for operating over sequences: doing the same thing to each element of the sequence.

Functions as values

Let's pause here for a second, because we're doing something unusual with functions. The `map` function takes a reference to a *function* as its first argument — not to the result of that function. When we wrote

```
map(sqrt, [1, 4, 9, 16])
```

we didn't *call* `sqrt` (like `sqrt(25)` is a call), instead we just used its name. In Python, the name of a function is a reference to an object representing that function. You can assign that object to another variable if you like, and it still behaves like `sqrt` :

```
>>> mySquareRoot = sqrt
>>> mySquareRoot(25)
5.0
```

You can also pass a reference to the function object as a parameter to another function, and that's what we're doing here with `map`. You can use function objects the same way you would use any other data value in Python (like numbers or strings or objects).

Functions are **first-class** in Python, meaning that they can be assigned to variables, passed as parameters, used as return values, and stored in data structures. First-class functions are a very powerful programming idea. The first practical programming language that used them was Lisp, invented by John McCarthy at MIT. But the idea of programming with functions as first-class values actually predates computers, tracing back to Alonzo Church's lambda calculus. The lambda calculus used the Greek letter λ to define new functions; this term stuck, and you'll see it as a keyword not only in Lisp and its descendants, but also in Python.

We've seen how to use built-in library functions as first-class values; how do we make our own? One way is using a familiar function definition, which gives the function a name:

```
>>> def powerOfTwo(k):
...     return 2**k
...
>>> powerOfTwo(5)
32
>>> map(powerOfTwo, [1, 2, 3, 4])
[2, 4, 8, 16]
```

When you only need the function in one place, however — which often comes up in programming with functions — it's more convenient to use a **lambda expression** :

```
lambda k: 2**k
```

This expression represents a function of one argument (called `k`) that returns the value 2^k . You can use it anywhere you would have used `powerOfTwo` :

```
>>> (lambda k: 2**k)(5)
32
>>> map(lambda k: 2**k, [1, 2, 3, 4])
[2, 4, 8, 16]
```

Python lambda expressions are unfortunately syntactically limited, to functions that can be written with just a `return` statement and nothing else (no `if` statements, no `for` loops, no local variables). But remember that's our goal with map/filter/reduce anyway, so it won't be a serious obstacle.

Guido Von Rossum, the creator of Python, wrote a blog post about the design principle that led not only to first-class functions in Python, but first-class methods as well: First-class Everything ([//python-history.blogspot.com/2009/02/first-class-everything.html](http://python-history.blogspot.com/2009/02/first-class-everything.html)) .

More ways to use map

Map is useful even if you don't care about the return value of the function. When you have a sequence of mutable objects, for example, you can map a mutator operation over them:

```
map(IOBase.close, streams) # closes each stream on the list
map(Thread.join, threads) # waits for each thread to finish
```

Some versions of map (including Python's built-in `map`) also support mapping functions with multiple arguments. For example, you can add two lists of numbers element-wise:

```
>>> import operator
>>> map(operator.add, [1, 2, 3], [4, 5, 6])
[5, 7, 9]
```

Filter

Our next important sequence operation is **filter**, which tests each element with a unary predicate. Elements that satisfy the predicate are kept; those that don't are removed. A new list is returned; filter doesn't modify its input list.

filter : (E → boolean) × Seq<E> → Seq<E>

Python examples:

```
>>> filter(str.isalpha, ['x', 'y', '2', '3', 'a'])
['x', 'y', 'a']
```

```
>>> def isOdd(x): return x % 2 == 1
...
>>> filter(isOdd, [1, 2, 3, 4])
[1, 3]
```

```
>>> filter(lambda s: len(s)>0, ['abc', '', 'd'])
['abc', 'd']
```

We can define filter in a straightforward way:

```
def filter(f, seq):
    result = []
    for elt in seq:
        if f(elt):
            result.append(elt)
    return result
```

Reduce

Our final operator, **reduce**, combines the elements of the sequence together, using a binary function. In addition to the function and the list, it also takes an *initial value* that initializes the reduction, and that ends up being the return value if the list is empty.

reduce : (F × E → F) × Seq<E> × F → F

`reduce(f, list, init)` combines the elements of the list from left to right, as follows:

```
result_0 = init
result_1 = f(result_0, list[0])
result_2 = f(result_1, list[1])
...
result_n = f(result_{n-1}, list[n-1])
```

`result_n` is the final result for an n -element list.

Adding numbers is probably the most straightforward example:

```
>>> reduce(lambda x,y: x+y, [1, 2, 3], 0)
6
# --or--
>>> import operator
>>> reduce(operator.add, [1, 2, 3], 0)
6
```

There are two design choices in the reduce operation. First is whether to require an initial value. In Python's reduce function, the initial value is optional, and if you omit it, reduce uses the first element of the list as its initial value. So you get behavior like this instead:

```
result_0 = undefined (reduce throws an exception if the list is empty)
result_1 = list[0]
result_2 = f(result_1, list[1])
...
result_n = f(result_{n-1}, list[n-1])
```

This makes it easier to use reducers like `max`, which have no well-defined initial value:

```
>>> reduce(max, [5, 8, 3, 1])
8
```

The second design choice is the order in which the elements are accumulated. For associative operators like `add` and `max` it makes no difference, but for other operators it can. Python's reduce is also called **fold-left** in other programming languages, because it combines the sequence starting from the left (the first element). **Fold-right** goes in the other direction:

fold-right : (E × F → F) × Seq<E> × F → F

where `fold-right(f, list, init)` of an n-element list follows this pattern:

```
result_0 = init
result_1 = f(list[n-1], result_0)
result_2 = f(list[n-2], result_1)
...
result_n = f(list[0], result_{n-1})
```

to produce `result_n` as the final result.

Here's a diagram of two ways to reduce: from the left or from the right:

fold-left : (F × E → F) × Seq<E> × F → F

fold-left(-, [1, 2, 3], 0) = -6



fold-right : (E × F → F) × Seq<E> × F → F

fold-right(-, [1, 2, 3], 0) = 2

The return type of the reduce operation doesn't have to match the type of the list elements. For example, we can use reduce to glue together a sequence into a string:

```
>>> reduce(lambda s,x: s+str(x), [1, 2, 3, 4], '')  
'1234'
```

Or to flatten out nested sublists into a single list:

```
>>> reduce(operator.concat, [[1, 2], [3, 4], [], [5]], [])  
[1, 2, 3, 4, 5]
```

This is a useful enough sequence operation that we'll define it as **flatten**, although it's just a reduce step inside:

```
def flatten(list):  
    return reduce(operator.concat, list, [])
```

More examples

Suppose we have a polynomial represented as a list of coefficients, $a[0], a[1], \dots, a[n-1]$, where $a[i]$ is the coefficient of x^i . Then we can evaluate it using map and reduce:

```
def evaluate(a, x):  
    xi = map(lambda i: x**i, range(0, len(a))) # [x^0, x^1, x^2, ..., x^(n-1)]  
    axi = map(operator.mul, a, xi)           # [a[0]*x^0, a[1]*x^1, ..., a[n-1]  
    *x^(n-1)]  
    return reduce(operator.add, axi, 0)       # sum of axi
```

This code uses the convenient Python generator method `range(a,b)`, which generates a list of integers from a to b-1. In map/filter/reduce programming, this kind of method replaces a `for` loop that indexes from a to b.

Now let's look at a typical database query example. Suppose we have a database about digital cameras, in which each object is of type `Camera` with observer methods for its properties (`brand()`, `pixels()`, `cost()`, etc.). The whole database is in a list called `cameras`. Then we can describe queries on this database using map/filter/reduce:

```
# What's the highest resolution Nikon sells?  
reduce(max, map(Camera.pixels, filter(lambda c: c.brand() == "Nikon", cameras)))
```

Relational databases use the map/filter/reduce paradigm (where it's called project/select/aggregate). SQL (<https://en.wikipedia.org/wiki/SQL>) (Structured Query Language) is the *de facto* standard language for querying relational databases. A typical SQL query looks like this:

```
select max(pixels) from cameras where brand = "Nikon"
```

`cameras` is a **sequence** (a list of rows, where each row has the data for one camera)

`where brand = "Nikon"` is a **filter**

`pixels` is a **map** (extracting just the pixels field from the row)

`max` is a **reduce**

Back to the intro example

Going back to the example we started with, where we want to find all the words in the Java files in our project, let's try creating a useful abstraction for filtering files by suffix:

```
def fileEndsWith(suffix):
    return lambda file: file.getName().endsWith(suffix)
```

`fileEndsWith` returns *functions* that are useful as filters: it takes a filename suffix like `.java` and dynamically generates a function that we can use with `filter` to test for that suffix:

```
filter(fileEndsWith(".java"), files)
```

`fileEndsWith` is a different kind of beast than our usual functions. It's a **higher-order function**, meaning that it's a function that takes another function as an argument, or returns another function as its result. Higher-order functions are operations on the datatype of functions; in this case, `fileEndsWith` is a *producer* of functions.

Now let's use `map`, `filter`, and `flatten` (which we defined above using `reduce`) to recursively traverse the folder tree:

```
def allFilesIn(folder):
    children = folder.listFiles()
    subfolders = filter(File.isDirectory, children)
    descendants = flatten(map(allFilesIn, subfolders))
    return descendants + filter(File.isFile, children)
```

The first line gets all the children of the folder, which might look like this:

```
["src/client", "src/server", "src/Main.java", ...]
```

The second line is the key bit: it filters the children for just the subfolders, and then recursively maps `allFilesIn` against this list of subfolders! The result might look like this:

```
[["src/client/MyClient.java", ...], ["src/server/MyServer.java", ...], ...]
```

So we have to flatten it to remove the nested structure. Then we add the immediate children that are plain files (not folders), and that's our result.

We can also do the other pieces of the problem with `map/filter/reduce`. Once we have the list of files we want to extract words from, we're ready to load their contents. We can use `map` to get their pathnames as strings, open them, and then read in each file as a list of files:

```
pathnames = map(File.getPath, files)
streams = map(open, pathnames)
lines = map(list, streams)
```

This actually looks like a single `map` operation where we want to apply three functions to the elements, so let's pause to create another useful higher-order function: composing functions together.

```
def compose(f, g):
    """Requires that f and g are functions, f:A->B and g:B->C.
    Returns a function A->C by composing f with g."""
    return lambda x: g(f(x))
```

Now we can use a single map:

```
lines = map(compose(compose(File.getPath, open), list), files)
```

Better, since we already have three functions to apply, let's design a way to compose an arbitrary chain of functions:

```
def chain(funcs):
    """Requires funcs is a list of functions [A->B, B->C, ..., Y->Z].
    Returns a fn A->Z that is the left-to-right composition of funcs."""
    return reduce(compose, funcs)
```

So that the map operation becomes:

```
lines = map(chain([File.getPath, open, list]), files)
```

Now we see more of the power of first-class functions. We can put functions into data structures and use operations on those data structures, like map, reduce, and filter, on the functions themselves!

Since this map will produce a list of lists of lines (one list of lines for each file), let's flatten it to get a single line list, ignoring file boundaries:

```
allLines = flatten(map(chain([File.getPath, open, list]), files))
```

Then we split each line into words similarly:

```
words = flatten(map(str.split, lines))
```

And we're done, we have our list of all words in the project's Java files! As promised, the control statements have disappeared.

→ **full Python code for the example** (<https://github.com/mit6005/sp16-ex25-words/blob/master/src/words/Words2.py>)

Benefits of abstracting out control

Map/filter/reduce can often make code shorter and simpler, and allow the programmer to focus on the heart of the computation rather than on the details of loops, branches, and control flow.

By arranging our program in terms of map, filter, and reduce, and in particular using immutable datatypes and pure functions (functions that do not mutate data) as much as possible, we've created more opportunities for safe concurrency. Maps and filters using pure functions over immutable datatypes are instantly parallelizable — invocations of the function on different elements of the sequence can be run in different threads, on different processors, even on different machines, and the result will still be the same. MapReduce (<https://en.wikipedia.org/wiki/MapReduce>) is a pattern for parallelizing large computations in this way.

First-class functions in Java

We've seen what first-class functions look like in Python; how does this all work in Java?

In Java, the only first-class values are primitive values (ints, booleans, characters, etc.) and object references. But objects can carry functions with them, in the form of methods. So it turns out that the way to implement a first-class function, in an object-oriented programming language like Java that doesn't support first-class functions directly, is to use an object with a method representing the function.

We've actually seen this before several times already:

- The `Runnable` object that you pass to a `Thread` constructor is a first-class function, `void run()`.
- The `Comparator<T>` object that you pass to a sorted collection (e.g. `SortedSet`) is a first-class function, `int compare(T o1, T o2)`.
- The `KeyListener` object that you register with the graphical user interface toolkit to get keyboard events is a bundle of several functions, `keyPressed(KeyEvent)`, `keyReleased(KeyEvent)`, etc.

This design pattern is called a **functional object** or **functor**, an object whose purpose is to represent a function.

Lambda expressions in Java

Java's lambda expression syntax provides a succinct way to create instances of functional objects. For example, instead of writing:

```
new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello!");
    }
}).start();
```

we can use a lambda expression:

```
new Thread(() -> {
    System.out.println("Hello");
}).start();
```

On the Java Tutorials page for Lambda Expressions, read **Syntax of Lambda Expressions** (<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax>).

There's no magic here: Java still doesn't have first-class functions. So you can only use a lambda when the Java compiler can verify two things:

1. It must be able to determine the type of the functional object the lambda will create. In this example, the compiler sees that the `Thread` constructor takes a `Runnable`, so it will infer that the type must be `Runnable`.
2. This inferred type must be *functional interface*: an interface with only one (abstract) method. In this example, `Runnable` indeed only has a single method — `void run()` — so the compiler knows the code in the body of the lambda belongs in the body of a `run` method of a new `Runnable` object.

Java provides some standard functional interfaces ([//docs.oracle.com/javase/8/docs/api/?java/util/function/package-summary.html](https://docs.oracle.com/javase/8/docs/api/?java/util/function/package-summary.html)) we can use to write code in the map/filter/reduce pattern, e.g.:

- `Function<T,R>` ([//docs.oracle.com/javase/8/docs/api/?java/util/function/Function.html](https://docs.oracle.com/javase/8/docs/api/?java/util/function/Function.html))
represents unary functions from `T` to `R`
- `BiFunction<T,U,R>` ([//docs.oracle.com/javase/8/docs/api/?java/util/function/BiFunction.html](https://docs.oracle.com/javase/8/docs/api/?java/util/function/BiFunction.html))
represents binary functions from `T × U` to `R`
- `Predicate<T>` ([//docs.oracle.com/javase/8/docs/api/?java/util/function/Predicate.html](https://docs.oracle.com/javase/8/docs/api/?java/util/function/Predicate.html))
represents functions from `T` to boolean

So we could implement map in Java like so:

```
/*
 * Apply a function to every element of a list.
 * @param f function to apply
 * @param list list to iterate over
 * @return [f(list[0]), f(list[1]), ..., f(list[n-1])]
 */
public static <T,R> List<R> map(Function<T,R> f, List<T> list) {
    List<R> result = new ArrayList<>();
    for (T t : list) {
        result.add(f.apply(t));
    }
    return result;
}
```

And here's an example of using map; first we'll write it using the familiar syntax:

```
// anonymous classes like this one are effectively lambda expressions
Function<String,String> toLowerCase = new Function<>() {
    public String apply(String s) { return s.toLowerCase(); }
};
map(toLowerCase, Arrays.asList(new String[] {"A", "b", "C"}));
```

And with a lambda expression:

```
map(s -> s.toLowerCase(), Arrays.asList(new String[] {"A", "b", "C"}));
// --or--
map((s) -> s.toLowerCase(), Arrays.asList(new String[] {"A", "b", "C"}));
// --or--
map((s) -> { return s.toLowerCase(); }, Arrays.asList(new String[] {"A", "b", "C"}));
```

In this example, the lambda expression is just wrapping a call to `String`'s `toLowerCase`. We can use a *method reference* to avoid writing the lambda, with the syntax `::`. The signature of the method we refer to must match the signature required by the functional interface for static typing to be satisfied:

```
map(String::toLowerCase, Arrays.asList(new String[] {"A", "b", "C"}));
```

In the Java Tutorials, you can read more about **method references**

(<https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html>) if you want the details.

Using a method reference (vs. calling it) in Java serves the same purpose as referring to a function by name (vs. calling it) in Python.

Map/filter/reduce in Java

The abstract sequence type we defined above exists in Java as `Stream`

([//docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html](https://docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html)) , which defines `map` , `filter` , `reduce` , and many other operations.

Collection types like `List` and `Set` provide a `stream()` operation that returns a `Stream` for the collection, and there's an `Arrays.stream` function for creating a `Stream` from an array.

Here's one implementation of `allFilesIn` in Java with map and filter:

```
public class Words {
    static Stream<File> allFilesIn(File folder) {
        File[] children = folder.listFiles();
        Stream<File> descendants = Arrays.stream(children)
            .filter(File::isDirectory)
            .flatMap(Words::allFilesIn);
        return Stream.concat(descendants,
            Arrays.stream(children).filter(File::isFile));
    }
}
```

The map-and-flatten pattern is so common that Java provides a `flatMap` operation to do just that, and we've used it instead of defining `flatten` .

Here's `endsWith` :

```
static Predicate<File> endsWith(String suffix) {
    return f -> f.getPath().endsWith(suffix);
}
```

Given a `Stream<File>` `files` , we can now write e.g. `files.filter(endsWith(".java"))` to obtain a new filtered stream.

Look at the **revised Java code for this example** (<https://github.com/mit6005/sp16-ex25-words/blob/master/src/words/Words3.java>) .

You can compare **all three versions** (<https://github.com/mit6005/sp16-ex25-words/tree/master/src/words>) : the familiar Java implementation, Python with map/filter/reduce, and Java with map/filter/reduce.

Higher-order functions in Java

`Map/filter/reduce` are of course higher-order functions; so is `endsWith` above. Let's look at two more that we saw before: `compose` and `chain` .

The `Function` interface provides `compose` — but the implementation is very straightforward. In particular, once you get the types of the arguments and return values correct, Java's static typing makes it pretty much impossible to get the method body wrong:

```
/**
 * Compose two functions.
 * @param f function A->B
 * @param g function B->C
 * @return new function A->C formed by composing f with g
 */
public static <A,B,C> Function<A,C> compose(Function<A,B> f,
                                              Function<B,C> g) {
    return t -> g.apply(f.apply(t));
    // --or--
    // return new Function<A,C>() {
    //     public C apply(A t) { return g.apply(f.apply(t)); }
    // };
}
```

It turns out that we *can't* write `chain` in strongly-typed Java, because `List`'s (and other collections) must be homogeneous — we can specify a list whose elements are all of type `Function<A,B>`, but not one whose first element is a `Function<A,B>`, second is a `Function<B,C>`, and so on.

But here's `chain` for functions of the same input/output type:

```
/**
 * Compose a chain of functions.
 * @param funcs list of functions A->A to compose
 * @return function A->A made by composing list[0] ... list[n-1]
 */
public static <A> Function<A,A> chain(List<Function<A,A>> funcs) {
    return funcs.stream().reduce(Function.identity(), Function::compose);
}
```

Our Python version didn't use an initial value in the `reduce`, it required a non-empty list of functions. In Java, we've provided the identity function (that is, $f(t) = t$) as the identity value for the reduction.

Summary

This reading is about modeling problems and implementing systems with *immutable data* and operations that implement *pure functions*, as opposed to *mutable data* and operations with *side effects*. *Functional programming* is the name for this style of programming.

Functional programming is much easier to do when you have *first-class functions* in your language and you can build *higher-order functions* that abstract away control flow code.

Some languages — Haskell ([//www.haskell.org/](http://www.haskell.org/)), Scala ([//www.scala-lang.org/](http://www.scala-lang.org/)), OCaml ([//ocaml.org/](http://ocaml.org/)) — are strongly associated with functional programming. Many other languages — JavaScript (<https://developer.mozilla.org/en-US/docs/JavaScript>), Swift (<https://developer.apple.com/swift/>), several ([//msdn.microsoft.com/en-us/library/67ef8sbd.aspx](http://msdn.microsoft.com/en-us/library/67ef8sbd.aspx)).NET ([//fsharp.org](http://fsharp.org)) languages ([//msdn.microsoft.com/en-us/vstudio/hh388573.aspx](http://msdn.microsoft.com/en-us/vstudio/hh388573.aspx)), Ruby ([//www.ruby-lang.org/](http://www.ruby-lang.org/)), and so on — use functional programming to a greater or lesser extent. With Java's recently-added functional language features, if you continue programming in Java you should expect to see more functional programming there, too.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 26: Little Languages

Representing code as data

Building languages to solve problems

Music language

To be continued

Reading 26: Little Languages

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

In this reading we will begin to explore the design of a **little language** for constructing and manipulating music. Here's the bottom line: when you need to solve a problem, instead of writing a *program* to solve just that one problem, build a *language* that can solve a range of related problems.

The goal for this reading is to introduce the idea of **representing code as data** and familiarize you with an initial version of the **music language**.

Representing code as data

Recall the `Formula` datatype from *Recursive Data Types* (./16-recursive-data-types/recursive/#another_example_boolean_formulas) :

```
Formula = Variable(name:String)
      + Not(formula:Formula)
      + And(left:Formula, right:Formula)
      + Or(left:Formula, right:Formula)
```

We used instances of `Formula` to take propositional logic formulas, e.g. $(p \vee q) \wedge (\neg p \vee r)$, and represent them in a data structure, e.g.:

```
And(Or(Variable("p"), Variable("q")),
    Or(Not(Variable("p")), Variable("r")))
```

In the parlance of grammars and parsers, formulas are a *language*, and `Formula` is an *abstract syntax tree* (./18-parser-generators).

But why did we define a `Formula` type? Java already has a way to represent expressions of Boolean variables with *logical and*, *or*, and *not*. For example, given boolean variables `p`, `q`, and `r`:

```
(p || q) && ((!p) || r)
```

Done!

The answer is that the Java code expression `(p || q) && ((!p) || r)` is evaluated as soon as we encounter it in our running program. The `Formula` value `And(Or(...), Or(...))` is a **first-class value** that can be stored, passed and returned from one method to another, manipulated, and evaluated now or later (or more than once) as needed.

The `Formula` type is an example of **representing code as data**, and we've seen many more.

Consider this functional object ([./25-map-filter-reduce/#first-class_functions_in_java](#)) :

```
class VariableNameComparator implements Comparator<Variable> {
    public int compare(Variable v1, Variable v2) {
        return v1.name().compareTo(v2.name());
    }
}
```

An instance of `VariableNameComparator` is a value that can be passed around, returned, and stored. But at any time, the function that it represents can be invoked by calling its `compare` method with a couple of `Variable` arguments:

```
Variable v1, v2;
Comparator<Variable> c = new VariableNameComparator();
...
int a = c.compare(v1, v2);
int b = c.compare(v2, v1);
SortedSet<Variable> vars = new TreeSet<>(c); // vars is sorted by name
```

Lambda expressions allow us to create functional objects with a compact syntax:

```
Comparator<Variable> c = (v1, v2) -> v1.name().compareTo(v2.name());
```

Building languages to solve problems

When we define an abstract data type ([./12-abstract-data-types/](#)), we're extending the universe of built-in types provided by Java to include a new type, with new operations, appropriate to our problem domain. This new type is like a new language: a new set of nouns (values) and verbs (operations) we can manipulate. Of course, those nouns and verbs are abstractions built on top the existing nouns and verbs which were themselves already abstractions.

A *language* has greater flexibility than a mere *program*, because we can use a language to solve a large class of related problems, instead of just a single problem.

- That's the difference between writing `(p || q) && ((!p) || r)` and devising a `Formula` type to represent the semantically-equivalent Boolean formula.
- And it's the difference between writing a matrix multiplication function and devising a `MatrixExpression` type ([./16-recursive-data-types/matexpr/](#)) to represent matrix multiplications — and store them, manipulate them, optimize them, evaluate them, and so on.

First-class functions and functional objects enable us to create particularly powerful languages because we can capture patterns of computation as reusable abstractions.

Music language

In class, we will design and implement a language for generating and playing music. To prepare, let's first understand the Java APIs for playing music with the MIDI (<https://en.wikipedia.org/wiki/MIDI>) synthesizer. We'll see how to write a *program* to play MIDI music. Then we'll begin to develop our music *language* by writing a recursive abstract data type for simple musical tunes. We'll choose a notation for writing music in strings, and we'll implement a parser to create instances of our `Music` type.

The full source code for the basic music language (<https://github.com/mit6005/sp16-ex26-music-starting>) is on GitHub.

Clone the `sp16-ex26-music-starting` (<https://github.com/mit6005/sp16-ex26-music-starting>) repo so you can run the code and follow the discussion below.

Playing MIDI music

`music.midi.MidiSequencePlayer` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/midi/MidiSequencePlayer.java>) uses the Java MIDI APIs to play sequences of notes. It's quite a bit of code, and you don't need to understand how it works.

`MidiSequencePlayer` implements the `music.SequencePlayer` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/SequencePlayer.java>) interface, allowing clients to use it without depending on the particular MIDI implementation. We *do* need to understand this interface and the types it depends on:

`addNote : SequencePlayer × Instrument × Pitch × double × double → void` ([SequencePlayer.java:15](https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/SequencePlayer.java#L15) (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/SequencePlayer.java#L15>)) is the workhorse of our music player. Calling this method schedules a musical pitch to be played at some time during the piece of music.

`play : SequencePlayer → void` ([SequencePlayer.java:20](https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/SequencePlayer.java#L20) (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/SequencePlayer.java#L20>)) actually plays the music. Until we call this method, we're just scheduling music that will, eventually, be played.

The `addNote` operation depends on two more types:

`Instrument` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Instrument.java>) is an enumeration of all the available MIDI instruments.

`Pitch` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Pitch.java>) is an abstract data type for musical pitches (think keys on the piano keyboard).

Read and understand the `Pitch` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Pitch.java>) documentation and the specifications for its `public constructor` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Pitch.java#L57-68>) and all its `public methods` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Pitch.java#L70-122>).

Our music data type will rely on `Pitch` in its rep, so be sure to understand the `Pitch` spec as well as its rep and abstraction function.

Using the MIDI sequence player and `Pitch`, we're ready to write code for our first bit of music!

Read and understand the `music.examples.ScaleSequence` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/examples/ScaleSequence.java>) code.

Run the main method in `ScaleSequence`. You should hear a one-octave scale!

Music data type

The `Pitch` datatype is useful, but if we want to represent a whole piece of music using `Pitch` objects, we should create an abstract data type to encapsulate that representation.

To start, we'll define the `Music` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Music.java>) type with a few operations:

`notes : String × Instrument → Music` (`MusicLanguage.java:51` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L51>)) makes a new `Music` from a string of simplified abc notation, described below.

`duration : Music → double` (`Music.java:11` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Music.java#L11>)) returns the duration, in beats, of the piece of music.

`play : Music × SequencePlayer × double → void` (`Music.java:18` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Music.java#L18>)) plays the piece of music using the given sequence player.

We'll implement `duration` and `play` as instance methods of `Music`, so we declare them in the `Music` interface.

`notes` will be a static factory method; rather than put it in `Music` (which we could do), we'll put it in a separate class: `MusicLanguage` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java>) will be our place for all the static methods we write to operate on `Music`.

Now that we've chosen some operations in the spec of `Music`, let's choose a representation.

- Looking at `ScaleSequence` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/examples/ScaleSequence.java>), the first concrete variant that might jump out at us is one to capture the information in each call to `addNote`: a particular pitch on a particular instrument played for some amount of time. We'll call this a `Note` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Note.java>).
- The other basic element of music is the silence between notes: `Rest` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Rest.java>).
- Finally, we need a way to glue these basic elements together into larger pieces of music. We'll choose a tree-like structure: `Concat(m1, m2:Music)` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Concat.java>) represents `m1` followed by `m2`, where `m1` and `m2` are any music.

This tree structure turns out to be an elegant decision as we further develop our `Music` type later on. In a real design process, we might iterate on the recursive structure of `Music` before we find the best implementation.

Here's the datatype definition:

```
Music = Note(duration:double, pitch:Pitch, instrument:Instrument)
      + Rest(duration:double)
      + Concat(m1:Music, m2:Music)
```

Composite

`Music` is an example of the **composite pattern**, in which we treat both single objects (*primitives*, e.g. `Note` and `Rest`) and groups of objects (*composites*, e.g. `Concat`) the same way.

- `Formula` is also an example of the composite pattern.
- The GUI view tree relies heavily on the composite pattern: there are *primitive views* like `JLabel` and `JTextField` that don't have children, and *composite views* like `JPanel` and `JScrollPane` that do contain other views as children. Both implement the common `JComponent` interface.

The composite pattern gives rise to a tree data structure, with primitives at the leaves and composites at the internal nodes.

Emptiness

One last design consideration: how do we represent the empty music? It's always good to have a representation for *nothing*, and we're certainly not going to use `null`.

We could introduce an `Empty` variant, but instead we'll use a `Rest` of duration `0` to represent emptiness.

Implementing basic operations

First we need to create the `Note` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Note.java>), `Rest` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Rest.java>), and `Concat` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Concat.java>) variants. All three are straightforward to implement, starting with constructors, `checkRep`, some observers, `toString`, and the equality methods.

- Since the `duration` operation is an instance method, each variant implements `duration` appropriately.
- The `play` operation is also an instance method; we'll discuss it below under *implementing the player*.

And we'll discuss the `notes` operation in *implementing the parser*.

Read and understand the `Note` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Note.java>), `Rest` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Rest.java>), and `Concat` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Concat.java>) classes.

To avoid representation exposure, let's add some additional static factory methods to the `Music` interface:

```
note : double × Pitch × Instrument → Music ( MusicLanguage.java:92
(https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L92)
)
```

```
rest : double → Music ( MusicLanguage.java:100 (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L100) )
```

`concat : Music × Music → Music` (`MusicLanguage.java:113` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L113>)) is our first producer operation.

All three of them are easy to implement by constructing the appropriate variant.

Music notation

We will write pieces of music using a simplified version of **abc notation** ([//en.wikipedia.org/wiki/ABC_notation](https://en.wikipedia.org/wiki/ABC_notation)) , a text-based music format.

We've already been representing pitches using their familiar letters. Our simplified abc notation represents sequences of **notes** and **rests** with syntax for indicating their **duration** , **accidental** (sharp or flat), and **octave** .

For example:

C D E F G A B C' B A G F E D C represents the one-octave ascending and descending C major scale we played in `ScaleSequence` . C is middle C, and C' is C one octave above middle C. Each note is a quarter note.

C/2 D/2 _E/2 F/2 G/2 _A/2 _B/2 C' is the ascending scale in C minor, played twice as fast. The E, A, and B are flat. Each note is an eighth note.

Read and understand the specification of **notes** in `MusicLanguage` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L20-50>) .

You don't need to understand the parser implementation yet, but you should understand the simplified abc notation enough to make sense of the examples.

If you're not familiar with music theory — why is an octave 8 notes but only 12 semitones? — don't worry. You might not be able to look at the abc strings and guess what they sound like, but you can understand the point of choosing a convenient textual syntax.

Implementing the parser

The `notes` method parses strings of simplified abc notation into `Music` .

notes : String × Instrument → Music (`MusicLanguage.java:51` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L51>)) splits the input into individual symbols (e.g. A,,/2 , .1/2). We start with the empty `Music` , `rest(0)` , symbols are parsed individually, and we build up the `Music` using `concat` .

parseSymbol : String × Instrument → Music (`MusicLanguage.java:62` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L62>)) returns a `Rest` or a `Note` for a single abc symbol (`symbol` in the grammar). It only parses the type (rest or note) and duration; it relies on `parsePitch` to handle pitch letters, accidentals, and octaves.

parsePitch : String → Pitch (`MusicLanguage.java:77` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L77>)) returns a `Pitch` by parsing a pitch grammar production. You should be able to understand the recursion — what's the base case? What are the recursive cases?

Implementing the player

Recall our operation for playing music:

play : Music × SequencePlayer × double → void (`Music.java:18` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Music.java#L18>)) plays the piece of music using the given sequence player after the given number of beats delay.

Why does this operation take `atBeat` ? Why not simply play the music now ?

If we define `play` in that way, we won't be able to play sequences of notes over time unless we actually `pause` during the `play` operation, for example with `Thread.sleep`. Our sequence player's `addNote` operation (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/SequencePlayer.java#L15>) is already designed to schedule notes in the future — it handles the delay.

With that design decision, it's straightforward to implement `play` in every variant of `Music`.

Read and understand the `Note.play` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Note.java#L55>), `Rest.play` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Rest.java#L33>), and `Concat.play` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Concat.java#L51>) methods.

You should be able to follow their recursive implementations.

Just one more piece of utility code before we're ready to jam: `music.midi.MusicPlayer` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/midi/MusicPlayer.java>) plays a `Music` using the `MidiSequencePlayer`. `Music` doesn't know about the concrete type of the sequence player, so we need a bit of code to bring them together.

Bringing this *all* together, let's use the `Music` ADT:

Read and understand the `music.examples.ScaleMusic` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/examples/ScaleMusic.java>) code.

Run the main method in `ScaleMusic`. You should hear the same one-octave scale again.

That's not very exciting, so **read `music.examples.RowYourBoatInitial`** (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/examples/RowYourBoatInitial.java>) and **run the main method**. You should hear *Row, row, row your boat!*

Can you follow the flow of the code from calling `notes(...)` to having an instance of `Music` to the recursive `play(...)` call to individual `addNote(...)` calls?

To be continued

Playing *Row, row, row your boat* is pretty exciting, but so far the most powerful thing we've done is not so much the *music language* as it is the very basic *music parser*. Writing music using the simplified abc notation is clearly much more **easy to understand**, **safe from bugs**, and **ready for change** than writing page after page of `addNote addNote addNote ...`

In class, we'll expand our music language and turn it into a powerful tool for constructing and manipulating complex musical structures.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 27: Team Version Control

Git workflow

Viewing commit history

Graph of commits

Using version control in a team

Team project

Reading 27: Team Version Control

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- Review Git basics and the commit graph
- Practice multi-user Git scenarios

Git workflow

You've been using Git for problem sets and in-class exercises for a while now. Most of the time, you haven't had to coordinate with other people pushing and pulling to and from the same repository as you at the same time. For the group projects, that will change.

In this reading, prepare for some in-class Git exercises by reviewing what you know and brushing up on some commands. Now that you're more comfortable with Git basics, it's a good time to go back and review some of the resources from the beginning of the semester.

Review **Inventing version control** ([..../05-version-control/#inventing_version_control](#)) : one developer, multiple developers, and branches.

If you need to, review **Learn the Git workflow** ([..../getting-started/#git](#)) from the *Getting Started* page.

Viewing commit history

Review **2.3 Viewing the Commit History** ([//git-scm.com/book/en/Git-Basics-Viewing-the-Commit-History](#)) from *Pro Git*.

You don't need to remember all the different command-line options presented in the book! Instead, learn what's possible so you know what to search for when you need it.

Clone the example repo from **Version Control** (`./05-version-control/#copy_an_object_graph_with_git_clone`) :
<https://github.com/mit6005/sp16-ex05-hello-git.git>

Use `log` commands to make sure you understand the history of the repo.

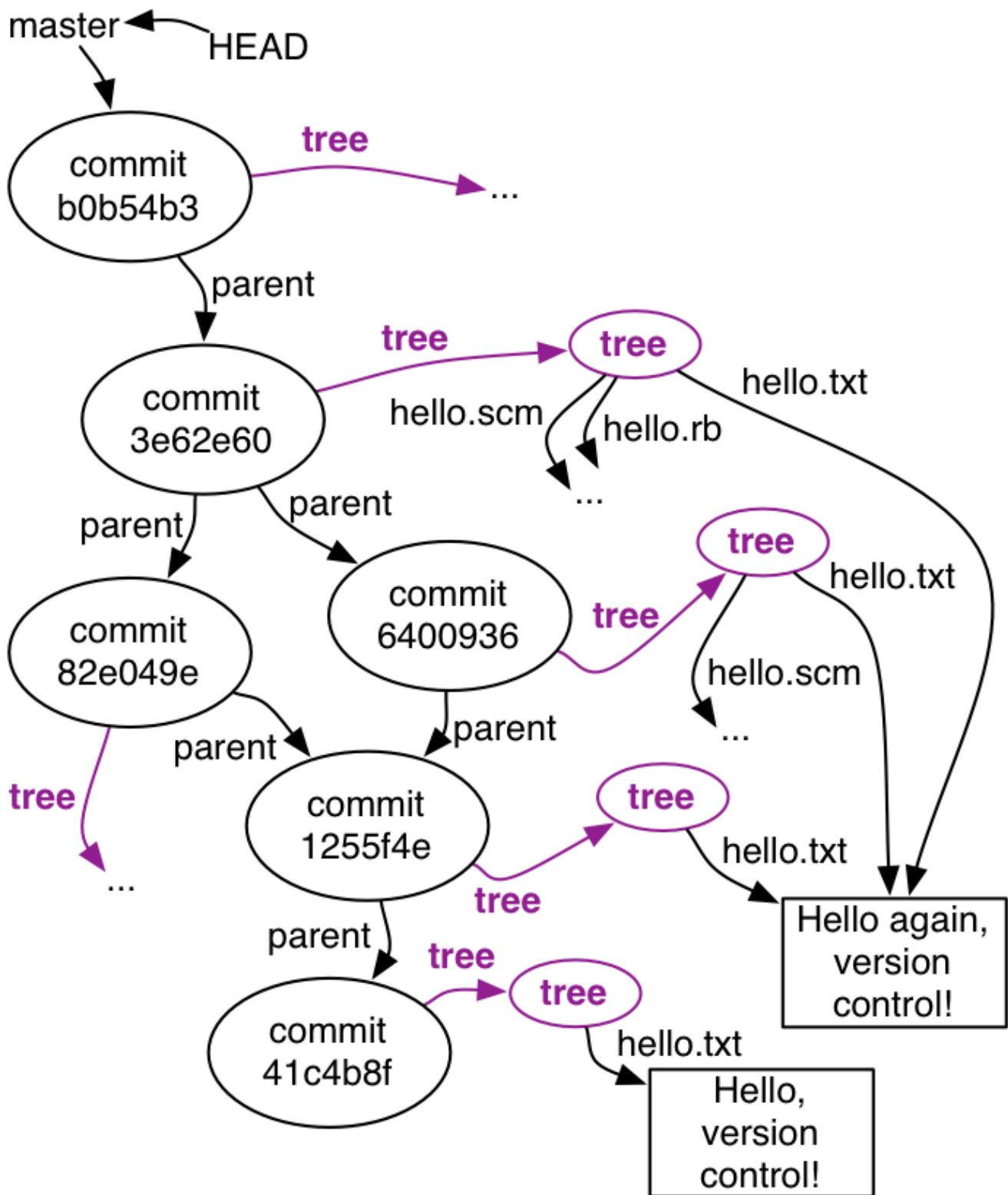
Graph of commits

Recall that the history recorded in a Git repository is a directed acyclic graph. The history of any particular branch in the repo (such as the default `master` branch) starts at some initial commit, and then its history may split apart and come back together, if multiple developers made changes in parallel (or if a single developer worked on two different machines without committing-pushing-pulling before the switch).

Here's the output of `git log` (`../../getting-started/#config-git`) for the example repository, which shows an ASCII-art graph:

```
* b0b54b3 (HEAD, origin/master, origin/HEAD, master) Greeting in Java
*   3e62e60 Merge
|\ 
| * 6400936 Greeting in Scheme
* | 82e049e Greeting in Ruby
|/
* 1255f4e Change the greeting
* 41c4b8f Initial commit
```

And here is a diagram of the DAG:



In the `ex05-hello-git` example repo, make sure you can explain where the history of `master` splits apart, and where it comes back together.

Review **Merging (./05-version-control/#merging)** from the *Version Control* reading.

You should understand every step of the process, and how it relates to the result in the example repo.

Review the **Getting Started section on merges (../../getting-started/#merges)**, including merging and merge conflicts.

Using version control in a team

Every team develops its own standards for version control, and the size of the team and the project they're working on is a major factor. Here are some guidelines for a small-scope team project of the kind you will undertake in 6.005:

- **Communicate.** Tell your teammates what you're going to work on. Tell them that you're working on it. And tell them that you worked on it. Communication is the best way to avoid wasted time and effort cleaning up broken code.
- **Write specs.** Necessary for the things we care about in 6.005, and part of good communication.
- **Write tests.** Don't wait for a giant pile of code to accumulate before you try to test it. Avoid having one person write tests while another person writes implementation (unless the implementation is a prototype you plan to throw away). Write tests first to make sure you agree on the specs. Everyone should take responsibility for the correctness of their code.
- **Run the tests.** Tests can't help you if you don't run them. Run them before you start working, run them again before you commit.
- **Automate.** You've already automated your tests with a tool like JUnit, but now you want to automate running those tests whenever the project changes. For 6.005 group projects, we provide Didit as a way to automatically run your tests every time a team member pushes to Athena. This also removes "it worked on my machine" from the equation: either it works in the automated build, or it needs to be fixed.
- **Review what you commit.** Use `git diff --staged` or a GUI program to see what you're about to commit. Run the tests. Don't use `commit -a`, that's a great way to fill your repo with `println`s and other stuff you didn't mean to commit. Don't annoy your teammates by committing code that doesn't compile, spews debug output, isn't actually used, etc.
- **Pull before you start working.** Otherwise, you probably don't have the latest version as your starting point — you're editing an old version of the code! You're guaranteed to have to merge your changes later, and you're in danger of having to waste time resolving a merge conflict.
- **Sync up.** At the end of a day or at the end of a work session, make sure everyone has pushed and pulled all the changes, you're all at the same commit, and everyone is satisfied with the state of the project.

We don't recommend using features like branching or rebasing for 6.005-sized projects.

We do strongly recommend working together in the same place at the same time, especially if this is your first group software engineering experience.

Team project

Reading 10: Recursion**Recursion****Choosing the Right Decomposition for a Problem****Structure of Recursive Implementations****Helper Methods****Choosing the Right Recursive Subproblem****Recursive Problems vs. Recursive Data****Reentrant Code****When to Use Recursion Rather Than Iteration****Common Mistakes in Recursive Implementations****Summary**

Reading 10: Recursion

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

After today's class, you should:

- be able to decompose a recursive problem into recursive steps and base cases
- know when and how to use helper methods in recursion
- understand the advantages and disadvantages of recursion vs. iteration

Recursion

In today's class, we're going to talk about how to implement a method, once you already have a specification. We'll focus on one particular technique, *recursion*. Recursion is not appropriate for every problem, but it's an important tool in your software development toolbox, and one that many people scratch their heads over. We want you to be comfortable and competent with recursion, because you will encounter it over and over. (That's a joke, but it's also true.)

Since you've taken 6.01, recursion is not completely new to you, and you have seen and written recursive functions like factorial and fibonacci before. Today's class will delve more deeply into recursion than you may have gone before. Comfort with recursive implementations will be necessary for upcoming classes.

A recursive function is defined in terms of *base cases* and *recursive steps*.

- In a base case, we compute the result immediately given the inputs to the function call.
- In a recursive step, we compute the result with the help of one or more *recursive calls* to this same function, but with the inputs somehow reduced in size or complexity, closer to a base case.

Consider writing a function to compute factorial. We can define factorial in two different ways:

Product	Recurrence relation
$n! = n \times (n - 1) \times \cdots \times 2 \times 1 = \prod_{k=1}^n k$ <p>(where the empty product equals multiplicative identity 1)</p>	$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n - 1)! \times n & \text{if } n > 0 \end{cases}$

which leads to two different implementations:

Iterative	Recursive
<pre>public static long factorial(int n) { long fact = 1; for (int i = 1; i <= n; i++) { fact = fact * i; } return fact; }</pre>	<pre>public static long factorial(int n) { if (n == 0) { return 1; } else { return n * factorial(n-1); } }</pre>

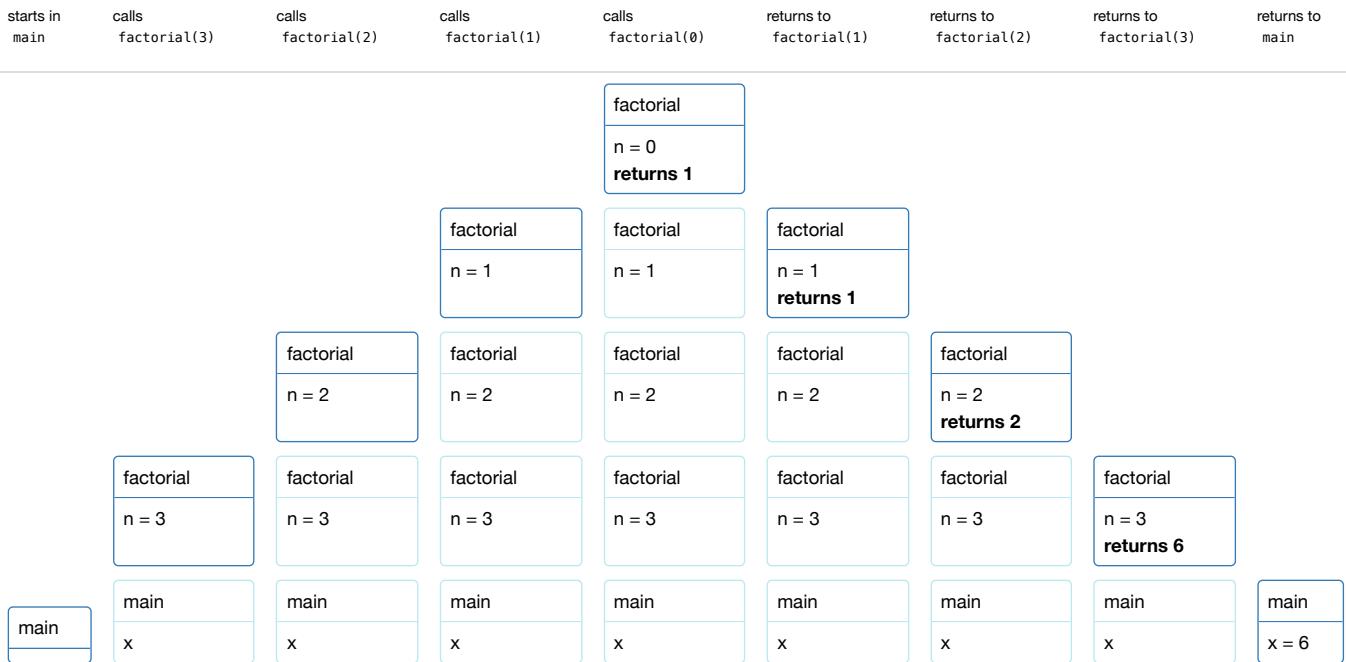
In the recursive implementation on the right, the base case is $n = 0$, where we compute and return the result immediately: $0!$ is defined to be 1. The recursive step is $n > 0$, where we compute the result with the help of a recursive call to obtain $(n-1)!$, then complete the computation by multiplying by n .

To visualize the execution of a recursive function, it is helpful to diagram the *call stack* of currently-executing functions as the computation proceeds.

Let's run the recursive implementation of factorial in a main method:

```
public static void main(String[] args) {
    long x = factorial(3);
}
```

At each step, with time moving left to right:



In the diagram, we can see how the stack grows as main calls factorial and factorial then calls *itself*, until factorial(0) does not make a recursive call. Then the call stack unwinds, each call to factorial returning its answer to the caller, until factorial(3) returns to main .

Here's an [interactive visualization of factorial](#)

([https://www.pythontutor.com/visualize.html#code=public+class+Factorial+%7B%0A++++public+static+void+main\(String%5B%5D+args\)+%7B%0A++++++long+x+%3D%20+n%3D%3D+0\)+%7B%0A++++++return+1%3B%0A++++++%7D+else+%7B%0A++++++return+n-*+factorial\(n-1\)%3B%0A++++++%7D%0A++++%7D%0A%7D&mode=display&origin=opt-frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=0](https://www.pythontutor.com/visualize.html#code=public+class+Factorial+%7B%0A++++public+static+void+main(String%5B%5D+args)+%7B%0A++++++long+x+%3D%20+n%3D%3D+0)+%7B%0A++++++return+1%3B%0A++++++%7D+else+%7B%0A++++++return+n-*+factorial(n-1)%3B%0A++++++%7D%0A++++%7D%0A%7D&mode=display&origin=opt-frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=0)) . You can step through the computation to see the recursion in action. New stack frames grow down instead of up in this visualization.

You've probably seen factorial before, because it's a common example for recursive functions. Another common example is the Fibonacci series:

```
/** 
 * @param n >= 0
 * @return the nth Fibonacci number
 */
public static int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return 1; // base cases
    } else {
        return fibonacci(n-1) + fibonacci(n-2); // recursive step
    }
}
```

Fibonacci is interesting because it has multiple base cases: n=0 and n=1. You can look at an [interactive visualization of Fibonacci](#)

([https://www.pythontutor.com/visualize.html#code=public+class+Fibonacci+%7B%0A++++public+static+void+main\(String%5B%5D+args\)+%7B%0A++++++int+x+%3D-\(n+%3D%3D+0+%7C%7C++%3D%3D+1\)+%7B%0A++++++return+1%3B%0A++++++%7D+else+%7B%0A++++++return+fibonacci\(n-1\)+%2B+fibonacci\(n-2\)%3B%0A++++++%7D%0A++++%7D%0A%7D&mode=display&origin=opt-frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=0](https://www.pythontutor.com/visualize.html#code=public+class+Fibonacci+%7B%0A++++public+static+void+main(String%5B%5D+args)+%7B%0A++++++int+x+%3D-(n+%3D%3D+0+%7C%7C++%3D%3D+1)+%7B%0A++++++return+1%3B%0A++++++%7D+else+%7B%0A++++++return+fibonacci(n-1)+%2B+fibonacci(n-2)%3B%0A++++++%7D%0A++++%7D%0A%7D&mode=display&origin=opt-frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=0)) . Notice that where factorial's stack steadily grows to a maximum depth and then shrinks back to the answer, Fibonacci's stack grows and shrinks repeatedly over the course of the computation.

Choosing the Right Decomposition for a Problem

Finding the right way to decompose a problem, such as a method implementation, is important. Good decompositions are simple, short, easy to understand, safe from bugs, and ready for change.

Recursion is an elegant and simple decomposition for some problems. Suppose we want to implement this specification:

```
/** 
 * @param word consisting only of letters A-Z or a-z
 * @return all subsequences of word, separated by commas,
 * where a subsequence is a string of letters found in word
 * in the same order that they appear in word.
 */
public static String subsequences(String word)
```

For example, `subsequences("abc")` might return `"abc,ab,bc,ac,a,b,c,"`. Note the trailing comma preceding the empty subsequence, which is also a valid subsequence.

This problem lends itself to an elegant recursive decomposition. Take the first letter of the word. We can form one set of subsequences that *include* that letter, and another set of subsequences that exclude that letter, and those two sets completely cover the set of possible subsequences.

```

1 public static String subsequences(String word) {
2     if (word.isEmpty()) {
3         return ""; // base case
4     } else {
5         char firstLetter = word.charAt(0);
6         String restOfWord = word.substring(1);
7
8         String subsequencesOfRest = subsequences(restOfWord);
9
10        String result = "";
11        for (String subsequence : subsequencesOfRest.split(", ", -1)) {
12            result += "," + subsequence;
13            result += "," + firstLetter + subsequence;
14        }
15        result = result.substring(1); // remove extra leading comma
16        return result;
17    }
18}

```

Structure of Recursive Implementations

A recursive implementation always has two parts:

- **base case**, which is the simplest, smallest instance of the problem, that can't be decomposed any further. Base cases often correspond to emptiness – the empty string, the empty list, the empty set, the empty tree, zero, etc.
- **recursive step**, which **decomposes** a larger instance of the problem into one or more simpler or smaller instances that can be solved by recursive calls, and then **recombines** the results of those subproblems to produce the solution to the original problem.

It's important for the recursive step to transform the problem instance into something smaller, otherwise the recursion may never end. If every recursive step shrinks the problem, and the base case lies at the bottom, then the recursion is guaranteed to be finite.

A recursive implementation may have more than one base case, or more than one recursive step. For example, the Fibonacci function has two base cases, $n=0$ and $n=1$.

Helper Methods

The recursive implementation we just saw for `subsequences()` is one possible recursive decomposition of the problem. We took a solution to a subproblem – the subsequences of the remainder of the string after removing the first character – and used it to construct solutions to the original problem, by taking each subsequence and adding the first character or omitting it. This is in a sense a *direct* recursive implementation, where we are using the existing specification of the recursive method to solve the subproblems.

In some cases, it's useful to require a stronger (or different) specification for the recursive steps, to make the recursive decomposition simpler or more elegant. In this case, what if we built up a partial subsequence using the initial letters of the word, and used the recursive calls to *complete* that partial subsequence using the remaining letters of the word? For example, suppose the original word is "orange". We'll both select "o" to be in the partial subsequence, and recursively extend it with all subsequences of "range"; and we'll skip "o", use "" as the partial subsequence, and again recursively extend it with all subsequences of "range".

Using this approach, our code now looks much simpler:

```

/**
 * Return all subsequences of word (as defined above) separated by commas,
 * with partialSubsequence prepended to each one.
 */
private static String subsequencesAfter(String partialSubsequence, String word) {
    if (word.isEmpty()) {
        // base case
        return partialSubsequence;
    } else {
        // recursive step
        return subsequencesAfter(partialSubsequence, word.substring(1))
            + ","
            + subsequencesAfter(partialSubsequence + word.charAt(0), word.substring(1));
    }
}

```

This `subsequencesAfter` method is called a **helper method**. It satisfies a different spec from the original `subsequences`, because it has a new parameter `partialSubsequence`. This parameter fills a similar role that a local variable would in an iterative implementation. It holds temporary state during the evolution of the computation. The recursive calls steadily extend this partial subsequence, selecting or ignoring each letter in the word, until finally reaching the end of the word (the base case), at which point the partial subsequence is returned as the only result. Then the recursion backtracks and fills in other possible subsequences.

To finish the implementation, we need to implement the original `subsequences` spec, which gets the ball rolling by calling the helper method with an initial value for the partial subsequence parameter:

```

public static String subsequences(String word) {
    return subsequencesAfter("", word);
}

```

Don't expose the helper method to your clients. Your decision to decompose the recursion this way instead of another way is entirely implementation-specific. In particular, if you discover that you need temporary variables like `partialSubsequence` in your recursion, don't change the original spec of your method, and don't force your clients to correctly initialize those parameters. That exposes your implementation to the client and reduces your ability to change it in the future. Use a private helper function for the recursion, and have your public method call it with the correct initializations, as shown above.

Choosing the Right Recursive Subproblem

Let's look at another example. Suppose we want to convert an integer to a string representation with a given base, following this spec:

```
/** 
 * @param n integer to convert to string
 * @param base base for the representation. Requires 2<=base<=10.
 * @return n represented as a string of digits in the specified base, with
 *         a minus sign if n<0.
 */
public static String stringValue(int n, int base)
```

For example, `stringValue(16, 10)` should return "16" , and `stringValue(16, 2)` should return "10000" .

Let's develop a recursive implementation of this method. One recursive step here is straightforward: we can handle negative integers simply by recursively calling for the representation of the corresponding positive integer:

```
if (n < 0) return "-" + stringValue(-n, base);
```

This shows that the recursive subproblem can be smaller or simpler in more subtle ways than just the value of a numeric parameter or the size of a string or list parameter. We have still effectively reduced the problem by reducing it to positive integers.

The next question is, given that we have a positive n, say n=829 in base 10, how should we decompose it into a recursive subproblem? Thinking about the number as we would write it down on paper, we could either start with 8 (the leftmost or highest-order digit), or 9 (the rightmost, lower-order digit). Starting at the left end seems natural, because that's the direction we write, but it's harder in this case, because we would need to first find the number of digits in the number to figure out how to extract the leftmost digit. Instead, a better way to decompose n is to take its remainder modulo base (which gives the *rightmost* digit) and also divide by base (which gives the subproblem, the remaining higher-order digits):

```
return stringValue(n/base, base) + "0123456789".charAt(n%base);
```

Think about several ways to break down the problem, and try to write the recursive steps. You want to find the one that produces the simplest, most natural recursive step.

It remains to figure out what the base case is, and include an if statement that distinguishes the base case from this recursive step.

Recursive Problems vs. Recursive Data

The examples we've seen so far have been cases where the problem structure lends itself naturally to a recursive definition. Factorial is easy to define in terms of smaller subproblems. Having a *recursive problem* like this is one cue that you should pull a recursive solution out of your toolbox.

Another cue is when the data you are operating on is inherently recursive in structure. We'll see many examples of recursive data a few classes from now, but for now let's look at the recursive data found in every laptop computer: its filesystem. A filesystem consists of named *files* . Some files are *folders* , which can contain other files. So a filesystem is recursive: folders contain other folders which contain other folders, until finally at the bottom of the recursion are plain (non-folder) files.

The Java library represents the file system using `java.io.File` ([//docs.oracle.com/javase/8/docs/api/index.html?java/io/File.html](https://docs.oracle.com/javase/8/docs/api/index.html?java/io/File.html)) . This is a recursive data type, in the sense that `f.getParentFile()` returns the parent folder of a file `f` , which is a `File` object as well, and `f.listFiles()` returns the files contained by `f` , which is an array of other `File` objects.

For recursive data, it's natural to write recursive implementations:

```
/** 
 * @param f a file in the filesystem
 * @return the full pathname of f from the root of the filesystem
 */
public static String fullPathname(File f) {
    if (f.getParentFile() == null) {
        // base case: f is at the root of the filesystem
        return f.getName();
    } else {
        // recursive step
        return fullPathname(f.getParentFile()) + "/" + f.getName();
    }
}
```

Recent versions of Java have added a new API, `java.nio.Files` ([//docs.oracle.com/javase/8/docs/api/index.html?java/nio/file/Files.html](https://docs.oracle.com/javase/8/docs/api/index.html?java/nio/file/Files.html)) and `java.nio.Path` ([//docs.oracle.com/javase/8/docs/api/index.html?java/nio/file/Path.html](https://docs.oracle.com/javase/8/docs/api/index.html?java/nio/file/Path.html)) , which offer a cleaner separation between the filesystem and the pathnames used to name files in it. But the data structure is still fundamentally recursive.

Reentrant Code

Recursion – a method calling itself – is a special case of a general phenomenon in programming called **reentrancy** . Reentrant code can be safely re-entered, meaning that it can be called again even while a *call to it is underway*. Reentrant code keeps its state entirely in parameters and local variables, and doesn't use static variables or global variables, and doesn't share aliases to mutable objects with other parts of the program, or other calls to itself.

Direct recursion is one way that reentrancy can happen. We've seen many examples of that during this reading. The `factorial()` method is designed so that `factorial(n-1)` can be called even though `factorial(n)` hasn't yet finished working.

Mutual recursion between two or more functions is another way this can happen – A calls B, which calls A again. Direct mutual recursion is virtually always intentional and designed by the programmer. But unexpected mutual recursion can lead to bugs.

When we talk about concurrency later in the course, reentrancy will come up again, since in a concurrent program, a method may be called at the same time by different parts of the program that are running concurrently.

It's good to design your code to be reentrant as much as possible. Reentrant code is safer from bugs and can be used in more situations, like concurrency, callbacks, or mutual recursion.

When to Use Recursion Rather Than Iteration

We've seen two common reasons for using recursion:

- The problem is naturally recursive (e.g. Fibonacci)
- The data is naturally recursive (e.g. filesystem)

Another reason to use recursion is to take more advantage of immutability. In an ideal recursive implementation, all variables are final, all data is immutable, and the recursive methods are all pure functions in the sense that they do not mutate anything. The behavior of a method can be understood simply as a relationship between its parameters and its return value, with no side effects on any other part of the program. This kind of paradigm is called *functional programming*, and it is far easier to reason about than *imperative programming* with loops and variables.

In iterative implementations, by contrast, you inevitably have non-final variables or mutable objects that are modified during the course of the iteration. Reasoning about the program then requires thinking about snapshots of the program state at various points in time, rather than thinking about pure input/output behavior.

One downside of recursion is that it may take more space than an iterative solution. Building up a stack of recursive calls consumes memory temporarily, and the stack is limited in size, which may become a limit on the size of the problem that your recursive implementation can solve.

Common Mistakes in Recursive Implementations

Here are two common ways that a recursive implementation can go wrong:

- The base case is missing entirely, or the problem needs more than one base case but not all the base cases are covered.
- The recursive step doesn't reduce to a smaller subproblem, so the recursion doesn't converge.

Look for these when you're debugging.

On the bright side, what would be an infinite loop in an iterative implementation usually becomes a `StackOverflowError` in a recursive implementation. A buggy recursive program fails faster.

Summary

We saw these ideas:

- recursive problems and recursive data
- comparing alternative decompositions of a recursive problem
- using helper methods to strengthen a recursive step
- recursion vs. iteration

The topics of today's reading connect to our three key properties of good software as follows:

- **Safe from bugs.** Recursive code is simpler and often uses immutable variables and immutable objects.
- **Easy to understand.** Recursive implementations for naturally recursive problems and recursive data are often shorter and easier to understand than iterative solutions.
- **Ready for change.** Recursive code is also naturally reentrant, which makes it safer from bugs and ready to use in more situations.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 11: Debugging

Reproduce the Bug

Understand the Location and Cause of the Bug

Fix the Bug

Reading 11: Debugging

6.005 Prime Objective

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

The topic of today's class is systematic debugging.

Sometimes you have no choice but to debug – particularly when the bug is found only when you plug the whole system together, or reported by a user after the system is deployed, in which case it may be hard to localize it to a particular module. For those situations, we can suggest a systematic strategy for more effective debugging.

Reproduce the Bug

Start by finding a small, repeatable test case that produces the failure. If the bug was found by regression testing, then you're in luck; you already have a failing test case in your test suite. If the bug was reported by a user, it may take some effort to reproduce the bug. For graphical user interfaces and multithreaded programs, a bug may be hard to reproduce consistently if it depends on timing of events or thread execution.

Nevertheless, any effort you put into making the test case small and repeatable will pay off, because you'll have to run it over and over while you search for the bug and develop a fix for it. Furthermore, after you've successfully fixed the bug, you'll want to add the test case to your regression test suite, so that the bug never crops up again. Once you have a test case for the bug, making this test work becomes your goal.

Here's an example. Suppose you have written this function:

```
/**  
 * Find the most common word in a string.  
 * @param text string containing zero or more words, where a word  
 *      is a string of alphanumeric characters bounded by nonalphanumeric.  
 * @return a word that occurs maximally often in text, ignoring alphabetic case.  
 */  
public static String mostCommonWord(String text) {  
    ...  
}
```

A user passes the whole text of Shakespeare's plays into your method, something like `mostCommonWord(allShakespearesPlaysConcatenated)`, and discovers that instead of returning a predictably common English word like "the" or "a", the method returns something unexpected, perhaps "e".

Shakespeare's plays have 100,000 lines containing over 800,000 words, so this input would be very painful to debug by normal methods, like print-debugging and breakpoint-debugging. Debugging will be easier if you first work on reducing the size of the buggy input to something manageable that still exhibits the same (or very similar) bug:

- does the first half of Shakespeare show the same bug? (Binary search! Always a good technique. More about this below.)
- does a single play have the same bug?
- does a single speech have the same bug?

Once you've found a small test case, find and fix the bug using that smaller test case, and then go back to the original buggy input and confirm that you fixed the same bug.

Understand the Location and Cause of the Bug

To localize the bug and its cause, you can use the scientific method:

1. **Study the data.** Look at the test input that causes the bug, and the incorrect results, failed assertions, and stack traces that result from it.
2. **Hypothesize.** Propose a hypothesis, consistent with all the data, about where the bug might be, or where it *cannot* be. It's good to make this hypothesis general at first.
3. **Experiment.** Devise an experiment that tests your hypothesis. It's good to make the experiment an *observation* at first – a probe that collects information but disturbs the system as little as possible.
4. **Repeat.** Add the data you collected from your experiment to what you knew before, and make a fresh hypothesis. Hopefully you have ruled out some possibilities and narrowed the set of possible locations and reasons for the bug.

Let's look at these steps in the context of the `mostCommonWord()` example, fleshed out a little more with three helper methods:

```

/**
 * Find the most common word in a string.
 * @param text string containing zero or more words,
 *             where a word is a string of alphanumeric
 *             characters bounded by nonalphanumeric.
 * @return a word that occurs maximally often in text,
 *         ignoring alphabetic case.
 */
public static String mostCommonWord(String text) {
    ... words = splitIntoWords(text); ...
    ... frequencies = countOccurrences(words); ...
    ... winner = findMostCommon(frequencies); ...
    ... return winner;
}

/** Split a string into words ... */
private static List<String> splitIntoWords(String text) {
    ...
}

/** Count how many times each word appears ... */
private static Map<String, Integer> countOccurrences(List<String> words) {
    ...
}

/** Find the word with the highest frequency count ... */
private static String findMostCommon(Map<String, Integer> frequencies) {
    ...
}

```

1. Study the Data

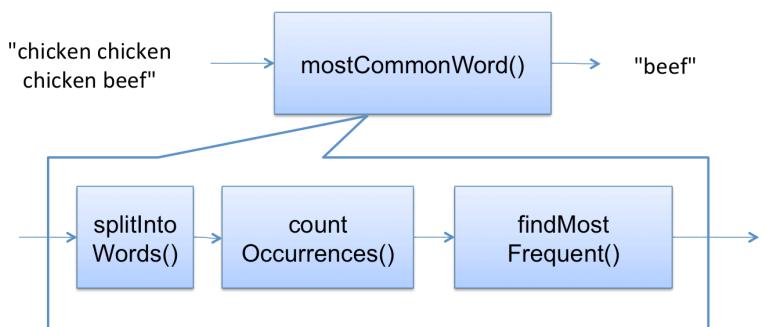
One important form of data is the stack trace from an exception. Practice reading the stack traces that you get, because they will give you enormous amounts of information about where and what the bug might be.

The process of isolating a small test case may also give you data that you didn't have before. You may even have two related test cases that *bracket* the bug in the sense that one succeeds and one fails. For example, maybe `mostCommonWords("c c, b")` is broken, but `mostCommonWords("c c b")` is fine.

2. Hypothesize

It helps to think about your program as modules, or steps in an algorithm, and try to rule out whole sections of the program at once.

The flow of data in `mostCommonWord()` is shown at right. If the symptom of the bug is an exception in `countOccurrences()`, then you can rule out everything downstream, specifically `findMostFrequent()`.



Then you would choose a hypothesis that tries to localize the bug even further. You might hypothesize that the bug is in `splitIntoWords()`, corrupting its results, which then cause the exception in `countOccurrences()`. You would then use an experiment to test that hypothesis. If the hypothesis is true, then you would have ruled out `countOccurrences()` as the source of the problem. If it's false, then you would rule out `splitIntoWords()`.

3. Experiment

A good experiment is a gentle observation of the system without disturbing it much. It might be:

- Run a **different test case**. The test case reduction process discussed above used test cases as experiments.
- Insert a **print statement** or **assertion** in the running program, to check something about its internal state.
- Set a **breakpoint** using a debugger, then single-step through the code and look at variable and object values.

It's tempting to try to insert *fixes* to the hypothesized bug, instead of mere probes. This is almost always the wrong thing to do. First, it leads to a kind of ad-hoc guess-and-test programming, which produces awful, complex, hard-to-understand code. Second, your fixes may just mask the true bug without actually removing it.

For example, if you're getting an `ArrayOutOfBoundsException`, try to understand what's going on first. Don't just add code that avoids or catches the exception, without fixing the real problem.

Other tips

Bug localization by binary search. Debugging is a search process, and you can sometimes use binary search to speed up the process. For example, in `mostCommonWords`, the data flows through three helper methods. To do a binary search, you would divide this workflow in half, perhaps guessing that the bug is found somewhere between the first helper method call and the second, and insert probes (like breakpoints, print statements, or assertions) there to check the results. From the answer to that experiment, you would further divide in half.

Prioritize your hypotheses. When making your hypothesis, you may want to keep in mind that different parts of the system have different likelihoods of failure. For example, old, well-tested code is probably more trustworthy than recently-added code. Java library code is probably more trustworthy than yours. The Java compiler and runtime, operating system platform, and hardware are increasingly more trustworthy, because they are more tried and tested. You should trust these lower levels until you've found good reason not to.

Swap components. If you have another implementation of a module that satisfies the same interface, and you suspect the module, then one experiment you can do is to try swapping in the alternative. For example, if you suspect your `binarySearch()` implementation, then substitute a simpler `linearSearch()` instead. If you suspect `java.util.ArrayList`, you could swap in `java.util.LinkedList` instead. If you suspect the Java runtime, run with a different version of Java. If you suspect the operating system, run your program on a different OS. If you suspect the hardware, run on a different machine. You can waste a lot of time swapping unfailing components, however, so don't do this unless you have good reason to suspect a component.

Make sure your source code and object code are up to date. Pull the latest version from the repository, and delete all your binary files and recompile everything (in Eclipse, this is done by Project → Clean).

Get help. It often helps to explain your problem to someone else, even if the person you're talking to has no idea what you're talking about. Lab assistants and fellow 6.005 students usually do know what you're talking about, so they're even better.

Sleep on it. If you're too tired, you won't be an effective debugger. Trade latency for efficiency.

Fix the Bug

Once you've found the bug and understand its cause, the third step is to devise a fix for it. Avoid the temptation to slap a patch on it and move on. Ask yourself whether the bug was a coding error, like a misspelled variable or interchanged method parameters, or a design error, like an underspecified or insufficient interface. Design errors may suggest that you step back and revisit your design, or at the very least consider all the other clients of the failing interface to see if they suffer from the bug too.

Think also whether the bug has any relatives. If I just found a divide-by-zero error here, did I do that anywhere else in the code? Try to make the code safe from future bugs like this. Also consider what effects your fix will have. Will it break any other code?

Finally, after you have applied your fix, add the bug's test case to your regression test suite, and run all the tests to assure yourself that (a) the bug is fixed, and (b) no new bugs have been introduced.

Summary

In this reading, we looked at how to debug systematically:

- reproduce the bug as a test case, and put it in your regression suite
- find the bug using the scientific method
- fix the bug thoughtfully, not slapdash

Thinking about our three main measures of code quality:

- **Safe from bugs.** We're trying to prevent them and get rid of them.
- **Easy to understand.** Techniques like static typing, final declarations, and assertions are additional documentation of the assumptions in your code. Variable scope minimization makes it easier for a reader to understand how the variable is used, because there's less code to look at.
- **Ready for change.** Assertions and static typing document the assumptions in an automatically-checkable way, so that when a future programmer changes the code, accidental violations of those assumptions are detected.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) | OCW 6.005
 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>)
 Spring 2016

Reading 12: Abstract Data Types

What Abstraction Means

Classifying Types and Operations

Designing an Abstract Type

Representation Independence

Realizing ADT Concepts in Java

Testing an Abstract Data Type

Summary

Reading 12: Abstract Data Types

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

Today's class introduces two ideas:

- Abstract data types
- Representation independence

In this reading, 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 problem: clients making assumptions about the type's internal 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.

Access Control in Java

You should already have read: **Controlling Access to Members of a Class** (<https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>) in the Java Tutorials.

What Abstraction Means

Abstract data types are an instance of a general principle in software engineering, which goes by many names with slightly different shades of meaning. Here are some of the names that are used for this idea:

- **Abstraction.** Omitting or hiding low-level details with a simpler, higher-level idea.
- **Modularity.** Dividing a system into components or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system.
- **Encapsulation.** Building walls around a module (a hard shell or capsule) so that the module is responsible for its own internal behavior, and bugs in other parts of the system can't damage its integrity.

- **Information hiding.** Hiding details of a module's implementation from the rest of the system, so that those details can be changed later without changing the rest of the system.
- **Separation of concerns.** Making a feature (or "concern") the responsibility of a single module, rather than spreading it across multiple modules.

As a software engineer, you should know these terms, because you will run into them frequently. The fundamental purpose of all of these ideas is to help achieve the three important properties that we care about in 6.005: safety from bugs, ease of understanding, and readiness for change.

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, e.g., 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 the original 6.170, the predecessor to 6.005. Barbara Liskov earned the Turing Award, computer science's equivalent of the Nobel Prize, for her work on abstract types.

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.

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 creator 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.
- **Observers** take objects of the abstract type and return objects of a different type. The `size` method of `List`, for example, returns an `int`.
- **Mutators** change objects. The `add` method of `List`, for example, mutates a list by adding an element to the end.

We can summarize these distinctions schematically like this (explanation to follow):

- $\text{creator} : t^* \rightarrow T$

- producer : $T^+ , t^* \rightarrow T$
- observer : $T^+ , t^* \rightarrow t$
- mutator : $T^+ , t^* \rightarrow \text{void} | t | T$

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. The $+$ marker indicates that the type may occur one or more times in that part of the signature, and the $*$ marker indicates that it occurs zero or more times. $|$ indicates or. For example, a producer may take two values of the abstract type T , like `String.concat()`

(<https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#concat-java.lang.String->) does:

- `concat : String × String → String`

Some observers take zero arguments of other types t , such as:

- `size (https://docs.oracle.com/javase/8/docs/api/java/util/List.html#size--) : List → int`

... and others take several:

- `regionMatches (https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#regionMatches-boolean-int-int) : String × boolean × int × String × int × int → boolean`

A creator operation is often implemented as a *constructor*, like `new ArrayList()`

(<https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#ArrayList->). But a creator can simply be a static method instead, like `Arrays.asList()` (<https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList-T-->). A creator implemented as a static method is often called a **factory method**. The various `String.valueOf()` (<https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#valueOf-boolean>) methods in Java are other examples of creators implemented as factory methods.

Mutators are often signaled by a `void` return type. A method that returns `void` *must* be called for some kind of side-effect, since otherwise it doesn't return anything. But not all mutators return `void`. For example, `Set.add()`

(<https://docs.oracle.com/javase/8/docs/api/java/util/Set.html#add-E>) returns a `boolean` that indicates whether the set was actually changed. In Java's graphical user interface toolkit, `Component.add()`

(<https://docs.oracle.com/javase/8/docs/api/java/awt/Container.html#add-javax.awt.Container>) returns the object itself, so that multiple `add()` calls can be chained together (https://en.wikipedia.org/wiki/Method_chaining).

Abstract Data Type Examples

Here are some examples of abstract data types, along with some of their operations, grouped by kind.

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 type. `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 `LinkedList`.

- creators: `ArrayList` and `LinkedList` constructors, `Collections.singletonList` (<https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#singletonList-T>)
- producers: `Collections.unmodifiableList` (<https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#unmodifiableList-T>)
- observers: `size` , `get`
- mutators: `add` , `remove` , `addAll` , `Collections.sort` (<https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#sort-java.util.List->)

String is Java's string type. `String` is immutable.

- creators: `String` 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 reserve the term *producer* only for operations that do no mutation.

Designing an Abstract Type

Designing an abstract type involves choosing good operations and determining how they should behave. Here are a few rules of thumb.

It's better to have **a few, simple operations** that can be combined in powerful ways, rather 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** in the sense that 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. For example, 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

Critically, 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 and postconditions, so that clients know what to depend on, and you know what you can safely change.

Example: Different Representations for Strings

Let's look at a simple abstract data type to see what representation independence means and why it's useful. The `MyString` type below has far fewer operations than the real Java `String`, and their specs are a little different, but it's still illustrative. Here are the specs for the ADT:

```
/** MyString represents an immutable sequence of characters. */
public class MyString {

    ///////////////////// Example of a creator operation ///////////////////
    /** @param b a boolean value
     *  @return string representation of b, either "true" or "false" */
    public static MyString valueOf(boolean b) { ... }

    ///////////////////// Examples of observer operations ///////////////////
    /** @return number of characters in this string */
    public int length() { ... }

    /** @param i character position (requires 0 <= i < string length)
     *  @return character at position i */
    public char charAt(int i) { ... }

    ///////////////////// Example of a producer operation ///////////////////
    /** Get the substring between start (inclusive) and end (exclusive).
     *  @param start starting index
     *  @param end ending index. Requires 0 <= start <= end <= string length.
     *  @return string consisting of charAt(start)...charAt(end-1) */
    public MyString substring(int start, int end) { ... }
}
```

These public operations and their specifications are the only information that a client of this data type is allowed to know. Following the test-first programming paradigm, in fact, the first client we should create is a test suite that exercises these operations according to their specs. At the moment, however, writing test cases that use `assertEquals` directly on `MyString` objects wouldn't work, because we don't have an equality operation defined on `MyString`. We'll talk about how to implement equality carefully in a later reading. For now, the only operations we can perform with `MyString`s are the ones we've defined above: `valueOf`, `length`, `charAt`, and `substring`. Our tests have to limit themselves to those operations. For example, here's one test for the `valueOf` operation:

```
MyString s = MyString.valueOf(true);
assertEquals(4, s.length());
assertEquals('t', s.charAt(0));
assertEquals('r', s.charAt(1));
assertEquals('u', s.charAt(2));
assertEquals('e', s.charAt(3));
```

We'll come back to the question of testing ADTs at the end of this reading.

For now, let's look at a simple representation for `MyString`: just an array of characters, exactly the length of the string, with no extra room at the end. Here's how that internal representation would be declared, as an instance variable within the class:

```
private char[] a;
```

With that choice of representation, the operations would be implemented in a straightforward way:

```

public static MyString valueOf(boolean b) {
    MyString s = new MyString();
    s.a = b ? new char[] { 't', 'r', 'u', 'e' }
             : new char[] { 'f', 'a', 'l', 's', 'e' };
    return s;
}

public int length() {
    return a.length;
}

public char charAt(int i) {
    return a[i];
}

public MyString substring(int start, int end) {
    MyString that = new MyString();
    that.a = new char[end - start];
    System.arraycopy(this.a, start, that.a, 0, end - start);
    return that;
}

```

Question to ponder: Why don't `charAt` and `substring` have to check whether their parameters are within the valid range? What do you think will happen if the client calls these implementations with illegal inputs?

One problem with this implementation is that it's passing up an opportunity for performance improvement. Because this data type is immutable, the `substring` operation doesn't really have to copy characters out into a fresh array. It could just point to the original `MyString` object's character array and keep track of the start and end that the new `substring` object represents. The `String` implementation in some versions of Java do this.

To implement this optimization, we could change the internal representation of this class to:

```

private char[] a;
private int start;
private int end;

```

With this new representation, the operations are now implemented like this:

```

public static MyString valueOf(boolean b) {
    MyString s = new MyString();
    s.a = b ? new char[] { 't', 'r', 'u', 'e' }
             : new char[] { 'f', 'a', 'l', 's', 'e' };
    s.start = 0;
    s.end = s.a.length;
    return s;
}

public int length() {
    return end - start;
}

public char charAt(int i) {
    return a[start + i];
}

public MyString substring(int start, int end) {
    MyString that = new MyString();
    that.a = this.a;
    that.start = this.start + start;
    that.end = this.start + end;
    return that;
}

```

Because `MyString`'s existing clients depend only on the specs of its public methods, not on its private fields, we can make this change without having to inspect and change all that client code. That's the power of representation independence.

Realizing ADT Concepts in Java

Let's summarize some of the general ideas we've discussed in this reading, which are applicable in general to programming in any language, and their specific realization using Java language features. The point is that there are several ways to do it, and it's important to both understand the big idea, like a creator operation, and different ways to achieve that idea in practice.

The only item in this table that hasn't yet been discussed in this reading is the use of a constant object as a creator operation. This pattern is commonly seen in immutable types, where the simplest or emptiest value of the type is simply a public constant, and producers are used to build up more complex values from it.

ADT concept	Ways to do it in Java	Examples
Creator operation	Constructor	<code>ArrayList()</code> https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#ArrayList--
	Static (factory) method	<code>Collections.singletonList()</code> https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#singletonList-T- , <code>Arrays.asList()</code> https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList-T...-
	Constant	<code>BigInteger.ZERO</code> https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html#ZERO
	Instance method	<code>List.get()</code> (https://docs.oracle.com/javase/8/docs/api/java/util/List.html#get-int-)
	Static method	<code>Collections.max()</code> https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#max-java.util.Collection-
Observer operation		

Producer operation	Instance method	<code>String.trim()</code> (//docs.oracle.com/javase/8/docs/api/java/lang/String.html#trim--)
	Static method	<code>Collections.unmodifiableList()</code> (//docs.oracle.com/javase/8/docs/api/java/util/Collections.html#unmodifiableList--java.util.List-)
Mutator operation	Instance method	<code>List.add()</code> (//docs.oracle.com/javase/8/docs/api/java/util/List.html#add-E-)
	Static method	<code>Collections.copy()</code> (//docs.oracle.com/javase/8/docs/api/java/util/Collections.html#copy--java.util.List-javax.util.List-)
Representation	private fields	

Testing an Abstract Data Type

We build a test suite for an abstract data type by creating tests for each of its operations. These tests inevitably interact with each other. The only way to test creators, producers, and mutators is by calling observers on the objects that result, and likewise, the only way to test observers is by creating objects for them to observe.

Here's how we might partition the input spaces of the four operations in our `MyString` type:

```
// testing strategy for each operation of MyString:
//
// valueOf():
//   true, false
// length():
//   string len = 0, 1, n
//   string = produced by valueOf(), produced by substring()
// charAt():
//   string len = 1, n
//   i = 0, middle, len-1
//   string = produced by valueOf(), produced by substring()
// substring():
//   string len = 0, 1, n
//   start = 0, middle, len
//   end = 0, middle, len
//   end-start = 0, n
//   string = produced by valueOf(), produced by substring()
```

Then a compact test suite that covers all these partitions might look like:

```

@Test public void testValueOfTrue() {
    MyString s = MyString.valueOf(true);
    assertEquals(4, s.length());
    assertEquals('t', s.charAt(0));
    assertEquals('r', s.charAt(1));
    assertEquals('u', s.charAt(2));
    assertEquals('e', s.charAt(3));
}

@Test public void testValueOfFalse() {
    MyString s = MyString.valueOf(false);
    assertEquals(5, s.length());
    assertEquals('f', s.charAt(0));
    assertEquals('a', s.charAt(1));
    assertEquals('l', s.charAt(2));
    assertEquals('s', s.charAt(3));
    assertEquals('e', s.charAt(4));
}

@Test public void testEndSubstring() {
    MyString s = MyString.valueOf(true).substring(2, 4);
    assertEquals(2, s.length());
    assertEquals('u', s.charAt(0));
    assertEquals('e', s.charAt(1));
}

@Test public void testMiddleSubstring() {
    MyString s = MyString.valueOf(false).substring(1, 2);
    assertEquals(1, s.length());
    assertEquals('a', s.charAt(0));
}

@Test public void testSubstringIsWholeString() {
    MyString s = MyString.valueOf(false).substring(0, 5);
    assertEquals(5, s.length());
    assertEquals('f', s.charAt(0));
    assertEquals('a', s.charAt(1));
    assertEquals('l', s.charAt(2));
    assertEquals('s', s.charAt(3));
    assertEquals('e', s.charAt(4));
}

@Test public void testSubstringOfEmptySubstring() {
    MyString s = MyString.valueOf(false).substring(1, 1).substring(0, 0);
    assertEquals(0, s.length());
}

```

Notice that each test case typically calls a few operations that *make* or *modify* objects of the type (creators, producers, mutators) and some operations that *inspect* objects of the type (observers). As a result, each test case covers parts of several operations.

Summary

- Abstract data types are characterized by their operations.
- Operations can be classified into creators, producers, observers, and mutators.
- An ADT's specification is its set of operations and their specs.
- A good ADT is simple, coherent, adequate, and representation-independent.
- An ADT is tested by generating tests for each of its operations, but using the creators, producers, mutators, and observers together in the same tests.

These ideas connect to our three key properties of good software as follows:

- **Safe from bugs.** A good ADT offers a well-defined contract for a data type, so that clients know what to expect from the data type, and implementors have well-defined freedom to vary.
- **Easy to understand.** A good ADT hides its implementation behind a set of simple operations, so that programmers using the ADT only need to understand the operations, not the details of the implementation.
- **Ready for change.** Representation independence allows the implementation of an abstract data type to change without requiring changes from its clients.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 13: Abstraction Functions & Rep Invariants

Invariants

Rep Invariant and Abstraction Function

Documenting the AF, RI, and Safety from Rep Exposure

ADT invariants replace preconditions

Summary

Reading 13: Abstraction Functions & Rep Invariants

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

Today's reading introduces several ideas:

- invariants
- representation exposure
- abstraction functions
- representation invariants

In this reading, we study a more formal mathematical idea of what it means for a class to implement an ADT, via the notions of *abstraction functions* and *rep invariants*. These mathematical notions are eminently practical in software design. The abstraction function will give us a way to cleanly define the equality operation on an abstract data type (which we'll discuss in more depth in a future class). The rep invariant will make it easier to catch bugs caused by a corrupted data structure.

Invariants

Resuming our discussion of what makes a good abstract data type, the final, and perhaps most important, property of a good abstract data type is that it **preserves its own invariants**. An *invariant* is a property of a program that is always true, for every possible runtime state of the program. 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. Saying that the ADT *preserves its own invariants* means that the ADT is responsible for ensuring that its own invariants hold. It doesn't depend on good behavior from its clients.

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 that uses Strings. Contrast that with a string type 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

Later in this reading, we'll see many interesting invariants. Let's focus on immutability for now. Here's a specific example:

```
/*
 * This immutable data type represents a tweet from Twitter.
 */
public class Tweet {

    public String author;
    public String text;
    public Date timestamp;

    /**
     * Make a Tweet.
     * @param author    Twitter user who wrote the tweet
     * @param text      text of the tweet
     * @param timestamp date/time when the tweet was sent
     */
    public Tweet(String author, String text, Date timestamp) {
        this.author = author;
        this.text = text;
        this.timestamp = timestamp;
    }
}
```

How do we guarantee that these Tweet objects are immutable – that, once a tweet is created, its author, message, and date 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:

```
Tweet t = new Tweet("justinbieber",
                     "Thanks to all those believers out there inspiring me every day",
                     new Date());
t.author = "rbmllr";
```

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 Tweet 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 Tweet {

    private final String author;
    private final String text;
    private final Date timestamp;

    public Tweet(String author, String text, Date timestamp) {
        this.author = author;
        this.text = text;
        this.timestamp = timestamp;
    }

    /** @return Twitter user who wrote the tweet */
    public String getAuthor() {
        return author;
    }

    /** @return text of the tweet */
    public String getText() {
        return text;
    }

    /** @return date/time when the tweet was sent */
    public Date getTimestamp() {
        return timestamp;
    }
}

```

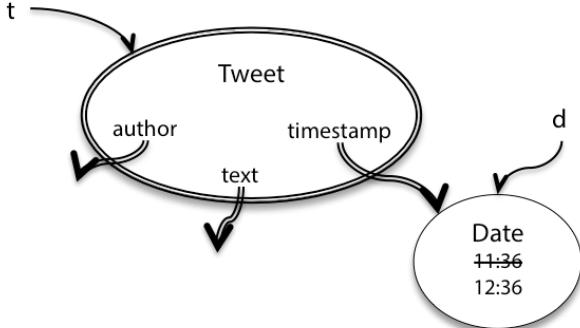
The `private` and `public` keywords indicate which fields and methods are accessible only within the class and which can be accessed from outside the class. The `final` keyword also helps by guaranteeing that the fields of this immutable type won't be reassigned after the object is constructed.

But that's not the end of the story: the rep is still exposed! Consider this perfectly reasonable client code that uses `Tweet` :

```

/** @return a tweet that retweets t, one
 * hour later*/
public static Tweet retweetLater(Tweet t)
{
    Date d = t.getTimestamp();
    d.setHours(d.getHours()+1);
    return new Tweet("rbmllr", t.getText
()), d);
}

```



`retweetLater` takes a tweet and should return another tweet with the same message (called a *retweet*) but sent an hour later. The `retweetLater` method might be part of a system that automatically echoes funny things that Twitter celebrities say.

What's the problem here? The `getTimestamp` call returns a reference to the same `Date` object referenced by tweet `t`. `t.timestamp` and `d` are aliases to the same mutable object. So when that date object is mutated by `d.setHours()`, this affects the date in `t` as well, as shown in the snapshot diagram.

`Tweet`'s immutability invariant has been broken. The problem is that `Tweet` leaked out a reference to a mutable object that its immutability depended on. We exposed the rep, in such a way that `Tweet` can no longer guarantee that its objects are immutable. Perfectly reasonable client code created a subtle bug.

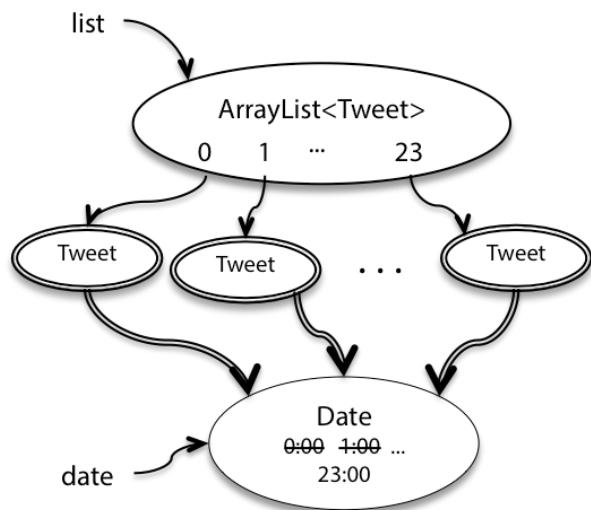
We can patch this kind of rep exposure by using defensive copying: making a copy of a mutable object to avoid leaking out references to the rep. Here's the code:

```
public Date getTimestamp() {
    return new Date(timestamp.getTime());
}
```

Mutable types often have a copy constructor that allows you to make a new instance that duplicates the value of an existing instance. In this case, `Date`'s copy constructor uses the timestamp value, measured in milliseconds since January 1, 1970. As another example, `StringBuilder`'s copy constructor takes a `String`. Another way to copy a mutable object is `clone()`, which is supported by some types but not all. There are unfortunate problems with the way `clone()` works in Java. For more, see Josh Bloch, *Effective Java* ([//library.mit.edu/item/001484188](https://library.mit.edu/item/001484188)), item 11.

So we've done some defensive copying in the return value of `getTimestamp`. But we're not done yet! There's still rep exposure. Consider this (again perfectly reasonable) client code:

```
/** @return a list of 24 inspiring tweets, one per hour today */
public static List<Tweet> tweetEveryHourToday () {
    List<Tweet> list = new ArrayList<Tweet>();
    Date date = new Date();
    for (int i = 0; i < 24; i++) {
        date.setHours(i);
        list.add(new Tweet("rbmllr", "keep it up! you can do it", date));
    }
    return list;
}
```



This code intends to advance a single `Date` object through the 24 hours of a day, creating a tweet for every hour. But notice that the constructor of `Tweet` saves the reference that was passed in, so all 24 `Tweet` objects end up with the same time, as shown in this snapshot diagram.

Again, the immutability of `Tweet` has been violated. We can fix this problem too by using judicious defensive copying, this time in the constructor:

```
public Tweet(String author, String text, Date timestamp) {
    this.author = author;
    this.text = text;
    this.timestamp = new Date(timestamp.getTime());
}
```

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. Doing that creates rep exposure.

You may object that this seems wasteful. Why make all these copies of dates? Why can't we just solve this problem by a carefully written specification, like this?

```
/**  
 * Make a Tweet.  
 * @param author    Twitter user who wrote the tweet  
 * @param text      text of the tweet  
 * @param timestamp date/time when the tweet was sent. Caller must never  
 *                   mutate this Date object again!  
 */  
public Tweet(String author, String text, Date timestamp) {
```

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 – as recommended in *Mutability & Immutability*'s Groundhog Day example ([..09-immutability/#risky_example_2_returning_mutable_values](#)) – we had used an immutable date object, like `java.time.ZonedDateTime`, instead of the mutable `java.util.Date`, then we would have ended this section after talking about `public` and `private`. No further rep exposure would have been possible.

Immutable Wrappers Around Mutable Data Types

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()`, etc. 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, as discussed in *Mutability & Immutability* ([..09-immutability/#useful_immutable_types](#))), 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. But that's still better than nothing, so using unmodifiable lists, maps, and sets can be a very good way to reduce the risk of bugs.

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 representation values (or rep values for short) 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 imaginations. 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.

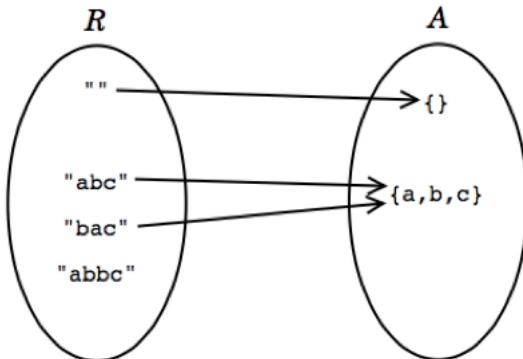
Now of course the implementor of the abstract type must be interested in the representation values, since it is the implementor's job to achieve the illusion of the abstract value space using the rep value space.

Suppose, for example, that we choose to use a string to represent a set of characters:

```
public class CharSet {
    private String s;
    ...
}
```

Then the rep space R contains Strings, and the abstract space A is mathematical sets of characters. 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 picture:

- **Every abstract value is mapped to by some rep value**. 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. In this case, we have decided that the string 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:

1. An *abstraction function* that maps rep values to the abstract values they represent:

```
AF : R → 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 surjective (also called *onto*), not necessarily injective (*one-to-one*) and therefore not necessarily bijective, and often partial.

2. A *rep invariant* that maps rep values to booleans:

```
RI : R → 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.

the abstract space and rep space of CharSet using the NoRepeatsRep

Both the rep invariant and the abstraction function should be documented in the code, right next to the declaration of the rep itself:

```
public class CharSet {
    private String s;
    // Rep invariant:
    //   s contains no repeated characters
    // Abstraction Function:
    //   represents the set of characters found in s
    ...
}
```

A common confusion about abstraction functions and rep invariants is 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.

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 different abstraction 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:

```
public class CharSet {
    private String s;
    // Rep invariant:
    //   s[0] <= s[1] <=
    ... <= s[s.length()-1]
    // Abstraction Function:
    //   represents the set of characters found in s
    ...
}
```

 the abstract space and rep space of CharSet using the SortedRep

Even with the same type for the rep value space and the same rep invariant RI, we might still interpret the rep differently, with different abstraction functions 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 rep "acgg" is interpreted as two range pairs, [a-c] and [g-g], and therefore represents the set {a,b,c,g}. Here's what the AF and RI would look like for that representation:

 the abstract space and rep space of CharSet using the SortedRangeRep

```
public class CharSet {  
    private String s;  
    // Rep invariant:  
    //   s.length is even  
    //   s[0] <= s[1] <= ... <= s[s.length()-1]  
    // Abstraction Function:  
    //   represents the union of the ranges  
    //   {s[i]...s[i+1]} for each adjacent pair of characters  
    //   in s  
    ...  
}
```

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**.

It's critically important to write down these assumptions in your code, as we've done above, so that future programmers (and your future self) are aware of what the representation actually means. Why? What happens if different implementers disagree about the meaning of the rep?

You can find example code for **three different CharSet implementations** (<https://github.com/mit6005/sp16-ex13-adt-examples/tree/master/src/charset>) on GitHub.

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.
     *  @param n value */
    public RatNum(int n) {
        numer = n;
        denom = 1;
        checkRep();
    }

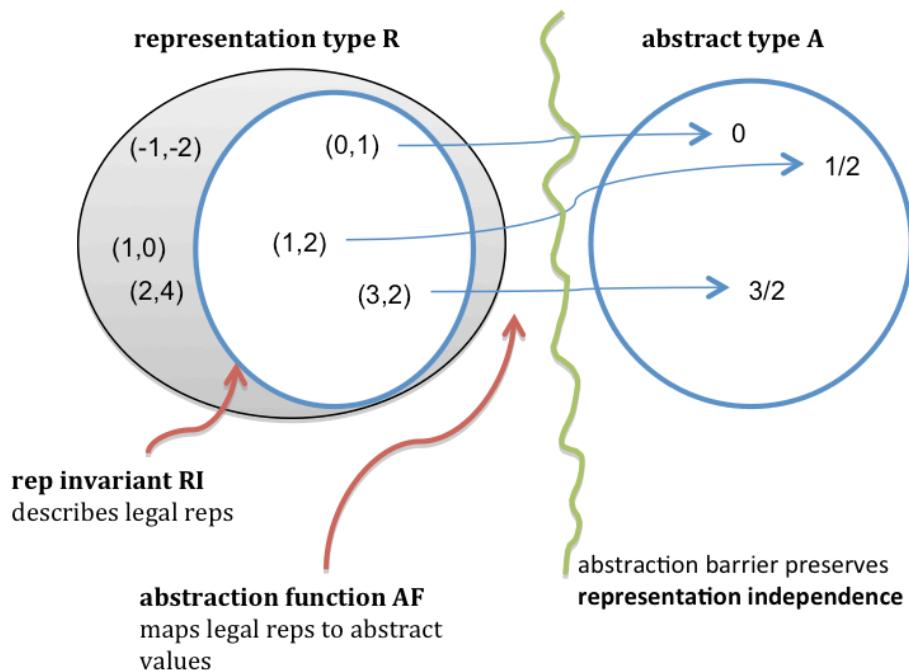
    /** Make a new RatNum == (n / d).
     *  @param n numerator
     *  @param d denominator
     *  @throws ArithmeticException if d == 0 */
    public RatNum(int n, int d) throws ArithmeticException {
        // reduce ratio to lowest terms
        int g = gcd(n, d);
        n = n / g;
        d = d / g;

        // make denominator positive
        if (d < 0) {
            numer = -n;
            denom = -d;
        } else {
            numer = n;
            denom = d;
        }
        checkRep();
    }
}

```

Here is a picture of the abstraction function and rep invariant for this code. The RI requires that numerator/denominator pairs be in reduced form (i.e., lowest terms), so pairs like (2,4) and (18,12) above should be drawn as outside the RI.

It would be completely reasonable to design another implementation of this same ADT with a more permissive RI. With such a change, some operations might become more expensive to perform, and others cheaper.



Checking the Rep Invariant

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. Here's a method for `RatNum` that tests its rep invariant:

```
// 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;
}
```

You should certainly call `checkRep()` to assert the rep invariant at the end of every operation that creates or mutates the rep – in other words, creators, producers, and mutators. Look back at the `RatNum` code above, and you'll see that it calls `checkRep()` at the end of both constructors.

Observer methods don't normally need to call `checkRep()`, but it's good defensive practice to do so anyway. Why? Calling `checkRep()` in every method, including observers, means you'll be more likely to catch rep invariant violations caused by rep exposure.

Why is `checkRep` private? Who should be responsible for checking and enforcing a rep invariant – clients, or the implementation itself?

No Null Values in the Rep

Recall from the *Specifications* reading ([..06-specifications/specs/](#)) that null values are troublesome and unsafe, so much so that we try to remove them from our programming entirely. In 6.005, the preconditions and postconditions of our methods implicitly require that objects and arrays be non-null.

We extend that prohibition to the reps of abstract data types. By default, in 6.005, the rep invariant implicitly includes `x != null` for every reference `x` in the rep that has object type (including references inside arrays or lists). So if your rep is:

```
class CharSet {
    String s;
}
```

then its rep invariant automatically includes `s != null`, and you don't need to state it in a rep invariant comment.

When it's time to implement that rep invariant in a `checkRep()` method, however, you still must *implement* the `s != null` check, and make sure that your `checkRep()` correctly fails when `s` is `null`. Often that check comes for free from Java, because checking other parts of your rep invariant will throw an exception if `s` is null. For example, if your `checkRep()` looks like this:

```
private void checkRep() {
    assert s.length() % 2 == 0;
    ...
}
```

then you don't need `assert s != null`, because the call to `s.length()` will fail just as effectively on a null reference. But if `s` is not otherwise checked by your rep invariant, then `assert s != null` explicitly.

Documenting the AF, RI, and Safety from Rep Exposure

It's good practice to document the abstraction function and rep invariant in the class, using comments right where the private fields of the rep are declared. We've been doing that above.

Another piece of documentation that 6.005 asks you to write is a **rep exposure safety argument**. This is a comment that examines each part of the rep, looks at the code that handles that part of the rep (particularly with respect to parameters and return values from clients, because that is where rep exposure occurs), and presents a reason why the code doesn't expose the rep.

Here's an example of `Tweet` with its rep invariant, abstraction function, and safety from rep exposure fully documented:

```
// Immutable type representing a tweet.
public class Tweet {

    private final String author;
    private final String text;
    private final Date timestamp;

    // Rep invariant:
    //   author is a Twitter username (a nonempty string of letters, digits, underscores)
    //   text.length <= 140
    // Abstraction Function:
    //   represents a tweet posted by author, with content text, at time timestamp

    // Safety from rep exposure:
    //   All fields are private;
    //   author and text are Strings, so are guaranteed immutable;
    //   timestamp is a mutable Date, so Tweet() constructor and getTimestamp()
    //       make defensive copies to avoid sharing the rep's Date object with clients.

    // Operations (specs and method bodies omitted to save space)
    public Tweet(String author, String text, Date timestamp) { ... }
    public String getAuthor() { ... }
    public String getText() { ... }
    public Date getTimestamp() { ... }
}
```

Notice that we don't have any explicit rep invariant conditions on `timestamp` (aside from the conventional assumption that `timestamp!=null`, which we have for all object references). But we still need to include `timestamp` in the rep exposure safety argument, because the immutability property of the whole type depends on all the fields remaining unchanged.

Compare the argument above with an example of a broken argument involving mutable `Date` objects (<https://github.com/mit6005/sp16-ex13-adt-examples/blob/master/src/Timespan.java>) .

Here are the arguments for `RatNum` .

```
// Immutable type representing a rational number.
public class RatNum {
    private final int numer;
    private final int denom;

    // Rep invariant:
    //   denom > 0
    //   numer/denom is in reduced form, i.e. gcd(|numer|,denom) = 1
    // Abstraction Function:
    //   represents the rational number numer / denom
    // Safety from rep exposure:
    //   All fields are private, and all types in the rep are immutable.

    // Operations (specs and method bodies omitted to save space)
    public RatNum(int n) { ... }
    public RatNum(int n, int d) throws ArithmeticException { ... }
    ...
}
```

Notice that an immutable rep is particularly easy to argue for safety from rep exposure.

You can find the full code for `RatNum` (<https://github.com/mit6005/sp16-ex13-adt-examples/blob/master/src/RatNum.java>) on GitHub.

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.

To make an invariant hold, we need to:

- make the invariant true in the initial state of the object; and
- ensure that all changes to the object keep the invariant true.

Translating this in terms of the types of ADT operations, this means:

- creators and producers must establish the invariant for new object instances; and
- mutators and observers must preserve the invariant.

The risk of rep exposure makes the situation more complicated. If the rep is exposed, then the object might be changed anywhere in the program, not just in the ADT's operations, and we can't guarantee that the invariant still holds after those arbitrary changes. So the full rule for proving invariants is:

Structural induction. If an invariant of an abstract data type is

1. established by creators and producers;
2. preserved by mutators, and observers; and
3. no representation exposure occurs,

then the invariant is true of all instances of the abstract data type.

ADT invariants replace preconditions

Now let's bring a lot of pieces together. An enormous advantage of a well-designed abstract data type is that it encapsulates and enforces properties that we would otherwise have to stipulate in a precondition. For example, instead of a spec like this, with an elaborate precondition:

```
/**  
 * @param set1 is a sorted set of characters with no repeats  
 * @param set2 is likewise  
 * @return characters that appear in one set but not the other,  
 * in sorted order with no repeats  
 */  
static String exclusiveOr(String set1, String set2);
```

We can instead use an ADT that captures the desired property:

```
/** @return characters that appear in one set but not the other */  
static SortedSet<Character> exclusiveOr(SortedSet<Character> set1, SortedSet<Character> set2);
```

This is easier to understand, because the name of the ADT conveys all the programmer needs to know. It's also safer from bugs, because Java static checking comes into play, and the required condition (sorted with no repeats) can be enforced in exactly one place, the `SortedSet` ([//docs.oracle.com/javase/8/docs/api/?java/util/SortedSet.html](https://docs.oracle.com/javase/8/docs/api/?java/util/SortedSet.html)) type.

Many of the places where we used preconditions on the problem sets would have benefited from a custom ADT instead.

Summary

- An invariant is a property that is always true of an ADT object instance, for the lifetime of the object.
- A good ADT preserves its own invariants. Invariants must be established by creators and producers, and preserved by observers and mutators.
- The rep invariant specifies legal values of the representation, and should be checked at runtime with `checkRep()`.
- The abstraction function maps a concrete representation to the abstract value it represents.
- Representation exposure threatens both representation independence and invariant preservation.

The topics of today's reading connect to our three properties of good software as follows:

- **Safe from bugs.** A good ADT preserves its own invariants, so that those invariants are less vulnerable to bugs in the ADT's clients, and violations of the invariants can be more easily isolated within the implementation of the ADT itself. Stating the rep invariant explicitly, and checking it at runtime with `checkRep()`, catches misunderstandings and bugs earlier, rather than continuing on with a corrupt data structure.
- **Easy to understand.** Rep invariants and abstraction functions explicate the meaning of a data type's representation, and how it relates to its abstraction.
- **Ready for change.** Abstract data types separate the abstraction from the concrete representation, which makes it possible to change the representation without having to change client code.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) | OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>)

Spring 2016

Reading 14: Interfaces

Interfaces

Subtypes

Example: MyString

Example: Set

Generic Interfaces

Why Interfaces?

Realizing ADT Concepts in Java

Summary

Reading 14: Interfaces

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

The topic of today's class is interfaces: separating the interface of an abstract data type from its implementation, and using Java `interface` types to enforce that separation.

After today's class, you should be able to define ADTs with interfaces, and write classes that implement interfaces.

Interfaces

Java's `interface` is a useful language mechanism for expressing an abstract data type. An interface in Java is a list of method signatures, but no method bodies. A class *implements* an interface if it declares the interface in its `implements` clause, and provides method bodies for all of the interface's methods. So one way to define an abstract data type in Java is as an interface, with its implementation as a class implementing that interface.

One advantage of this approach is that the interface specifies the contract for the client and nothing more. The interface is all a client programmer needs to read to understand the ADT. The client can't create inadvertent dependencies on the ADT's rep, because instance variables can't be put in an interface at all. The implementation is kept well and truly separated, in a different class altogether.

Another advantage is that multiple different representations of the abstract data type can co-exist in the same program, as different classes implementing the interface. When an abstract data type is represented just as a single class, without an interface, it's harder to have multiple representations. In the `MyString` example from *Abstract Data Types* ([./12-abstract-data-types/#example_different_representations_for_strings](#)) , `MyString` was a single class. We explored two different representations for `MyString` , but we couldn't have both representations for the ADT in the same program.

Java's static type checking allows the compiler to catch many mistakes in implementing an ADT's contract. For instance, it is a compile-time error to omit one of the required methods, or to give a method the wrong return type. Unfortunately, the compiler doesn't check for us that the code adheres to the specs of those methods that are

written in documentation comments.

For the details of how to define interfaces in Java, consult the **Java Tutorials section on interfaces** ([//docs.oracle.com/javase/tutorial/java/landl/createinterface.html](https://docs.oracle.com/javase/tutorial/java/landl/createinterface.html)) .

Subtypes

Recall that a *type* is a set of values. The Java `List` ([//docs.oracle.com/javase/8/docs/api/?java/util/List.html](https://docs.oracle.com/javase/8/docs/api/?java/util/List.html)) type is defined by an interface. If we think about all possible `List` values, none of them are `List` objects: we cannot create instances of an interface. Instead, those values are all `ArrayList` objects, or `LinkedList` objects, or objects of another class that implements `List`. A *subtype* is simply a subset of the *supertype*: `ArrayList` and `LinkedList` are subtypes of `List`.

“B is a subtype of A” means “every B is an A.” In terms of specifications: “every B satisfies the specification for A.”

That means B is only a subtype of A if B’s specification is at least as strong as A’s specification. When we declare a class that implements an interface, the Java compiler enforces part of this requirement automatically: for example, it ensures that every method in A appears in B, with a compatible type signature. Class B cannot implement interface A without implementing all of the methods declared in A.

But the compiler cannot check that we haven’t weakened the specification in other ways: strengthening the precondition on some inputs to a method, weakening a postcondition, weakening a guarantee that the interface abstract type advertises to clients. If you declare a subtype in Java — implementing an interface is our current focus — then you must ensure that the subtype’s spec is at least as strong as the supertype’s.

Example: MyString

Let’s revisit `MyString` ([./12-abstract-data-types/#example_different_representations_for_strings](#)). Using an interface instead of a class for the ADT, we can support multiple implementations:

```
/** MyString represents an immutable sequence of characters. */
public interface MyString {

    // We'll skip this creator operation for now
    // /** @param b a boolean value
    //  * @return string representation of b, either "true" or "false" */
    // public static MyString valueOf(boolean b) { ... }

    /** @return number of characters in this string */
    public int length();

    /** @param i character position (requires 0 <= i < string length)
     * @return character at position i */
    public char charAt(int i);

    /** Get the substring between start (inclusive) and end (exclusive).
     * @param start starting index
     * @param end ending index. Requires 0 <= start <= end <= string length.
     * @return string consisting of charAt(start)...charAt(end-1) */
    public MyString substring(int start, int end);
}
```

We’ll skip the static `valueOf` method and come back to it in a minute. Instead, let’s go ahead using a different technique from our toolbox of ADT concepts in Java ([./12-abstract-data-types/#realizing_adt_concepts_in_java](#)): constructors.

Here’s our first implementation:

```

public class SimpleMyString implements MyString {

    private char[] a;

    /* Create an uninitialized SimpleMyString. */
    private SimpleMyString() {}

    /** Create a string representation of b, either "true" or "false".
     * @param b a boolean value */
    public SimpleMyString(boolean b) {
        a = b ? new char[] { 't', 'r', 'u', 'e' }
              : new char[] { 'f', 'a', 'l', 's', 'e' };
    }

    @Override public int length() { return a.length; }

    @Override public char charAt(int i) { return a[i]; }

    @Override public MyString substring(int start, int end) {
        SimpleMyString that = new SimpleMyString();
        that.a = new char[end - start];
        System.arraycopy(this.a, start, that.a, 0, end - start);
        return that;
    }
}

```

And here's the optimized implementation:

```

public class FastMyString implements MyString {

    private char[] a;
    private int start;
    private int end;

    /* Create an uninitialized FastMyString. */
    private FastMyString() {}

    /** Create a string representation of b, either "true" or "false".
     * @param b a boolean value */
    public FastMyString(boolean b) {
        a = b ? new char[] { 't', 'r', 'u', 'e' }
              : new char[] { 'f', 'a', 'l', 's', 'e' };
        start = 0;
        end = a.length;
    }

    @Override public int length() { return end - start; }

    @Override public char charAt(int i) { return a[start + i]; }

    @Override public MyString substring(int start, int end) {
        FastMyString that = new FastMyString();
        that.a = this.a;
        that.start = this.start + start;
        that.end = this.start + end;
        return that;
    }
}

```

- Compare these classes to the implementations of `MyString` in *Abstract Data Types* (./12-abstract-data-types/#example_different_representations_for_strings). Notice how the code that previously appeared in static `valueOf` methods now appears in the constructors, slightly changed to refer to the rep of `this`.
- Also notice the use of `@Override` (<https://docs.oracle.com/javase/tutorial/java/annotations/predefined.html>). This annotation informs the compiler that the method must have the same signature as one of the methods in the interface we're implementing. But since the compiler already checks that we've implemented all of the interface methods, the primary value of `@Override` here is for readers of the code: it tells us to look for the spec of that method in the interface. Repeating the spec wouldn't be DRY, but saying nothing at all makes the code harder to understand.
- And notice the private empty constructors we use to make new instances in `substring(..)` before we fill in their reps with data. We didn't have to write these empty constructors before because Java provides them by default when we don't declare any others. Adding the constructors that take `boolean b` means we have to declare the empty constructors explicitly.

Now that we know good ADTs scrupulously preserve their own invariants (./13-abstraction-functions-rep-invariants/#invariants), these do-nothing constructors are a **bad** pattern: they don't assign any values to the rep, and they certainly don't establish any invariants. We should strongly consider revising the implementation. Since `MyString` is immutable, a starting point would be making all the fields `final`.

How will clients use this ADT? Here's an example:

```
MyString s = new FastMyString(true);
System.out.println("The first character is: " + s.charAt(0));
```

This code looks very similar to the code we write to use the Java collections classes:

```
List<String> s = new ArrayList<String>();
...
```

Unfortunately, this pattern **breaks the abstraction barrier** we've worked so hard to build between the abstract type and its concrete representations. Clients must know the name of the concrete representation class. Because interfaces in Java cannot contain constructors, they must directly call one of the concrete class' constructors. The spec of that constructor won't appear anywhere in the interface, so there's no static guarantee that different implementations will even provide the same constructors.

Fortunately, (as of Java 8) interfaces are allowed to contain static methods, so we can implement the creator operation `valueOf` as a static factory method in the interface `MyString`:

```
public interface MyString {

    /** @param b a boolean value
     *  @return string representation of b, either "true" or "false" */
    public static MyString valueOf(boolean b) {
        return new FastMyString(true);
    }

    // ...
}
```

Now a client can use the ADT without breaking the abstraction barrier:

```
MyString s = MyString.valueOf(true);
System.out.println("The first character is: " + s.charAt(0));
```

Example: Set

Java's collection classes provide a good example of the idea of separating interface and implementation.

Let's consider as an example one of the ADTs from the Java collections library, `Set`. `Set` is the ADT of finite sets of elements of some other type `E`. Here is a simplified version of the `Set` interface:

```
/** A mutable set.
 * @param <E> type of elements in the set */
public interface Set<E> {
```

`Set` is an example of a *generic type*: a type whose specification is in terms of a placeholder type to be filled in later. Instead of writing separate specifications and implementations for `Set<String>`, `Set<Integer>`, and so on, we design and implement one `Set<E>`.

We can match Java interfaces with our classification of ADT operations, starting with a creator:

```
// example creator operation
/** Make an empty set.
 * @param <E> type of elements in the set
 * @return a new set instance, initially empty */
public static <E> Set<E> make() { ... }
```

The `make` operation is implemented as a static factory method. Clients will write code like:

```
Set<String> strings = Set.make();
```

and the compiler will understand that the new `Set` is a set of `String` objects.

```
// example observer operations

/** Get size of the set.
 * @return the number of elements in this set */
public int size();

/** Test for membership.
 * @param e an element
 * @return true iff this set contains e */
public boolean contains(E e);
```

Next we have two observer methods. Notice how the specs are in terms of our abstract notion of a set; it would be malformed to mention the details of any particular implementation of sets with particular private fields. These specs should apply to any valid implementation of the set ADT.

```
// example mutator operations

/** Modifies this set by adding e to the set.
 * @param e element to add */
public void add(E e);

/** Modifies this set by removing e, if found.
 * If e is not found in the set, has no effect.
 * @param e element to remove */
public void remove(E e);
```

The story for these mutators is basically the same as for the observers. We still write specs at the level of our abstract model of sets.

In the Java Tutorials, read these pages:

- **Lesson: Interfaces** ([//docs.oracle.com/javase/tutorial/collections/interfaces/](https://docs.oracle.com/javase/tutorial/collections/interfaces/))
- **The Set Interface** ([//docs.oracle.com/javase/tutorial/collections/interfaces/set.html](https://docs.oracle.com/javase/tutorial/collections/interfaces/set.html))
- **Set Implementations** ([//docs.oracle.com/javase/tutorial/collections/implementations/set.html](https://docs.oracle.com/javase/tutorial/collections/implementations/set.html))
- **The List Interface** ([//docs.oracle.com/javase/tutorial/collections/interfaces/list.html](https://docs.oracle.com/javase/tutorial/collections/interfaces/list.html))

- [List Implementations](https://docs.oracle.com/javase/tutorial/collections/implementations/list.html) ([//docs.oracle.com/javase/tutorial/collections/implementations/list.html](https://docs.oracle.com/javase/tutorial/collections/implementations/list.html))

Generic Interfaces

Suppose we want to implement the generic `Set<E>` interface above.

Generic interface, non-generic implementation. One way we might do this is to implement `Set<E>` for a particular type `E`.

In *Abstraction Functions & Rep Invariants* ([./13-abstraction-functions-rep-invariants/#rep_invariant_and_abstraction_function](#)) we looked at `CharSet`, which represents a set of characters. The example code for `CharSet` (<https://github.com/mit6005/sp16-ex13-adt-examples/tree/master/src/charset>) includes a generic `Set` interface (<https://github.com/mit6005/sp16-ex13-adt-examples/blob/master/src/charset/Set.java>) and each of the implementations `CharSet1 / 2 / 3` declare:

```
public class CharSet implements Set<Character>
```

When the interface mentions placeholder type `E`, the `CharSet` implementations replace `E` with `Character`. For example:

```
public interface Set<E> {
    // ...
    /**
     * Test for membership.
     * @param e an element
     * @return true iff this set contains e
     */
    public boolean contains(E e);

    /**
     * Modifies this set by adding e to the
     * set.
     * @param e element to add
     */
    public void add(E e);
    // ...
}
```

```
public class CharSet1 implements Set<Character> {
    private String s = "";

    // ...

    @Override
    public boolean contains(Character e) {
        checkRep();
        return s.indexOf(e) != -1;
    }

    @Override
    public void add(Character e) {
        if (!contains(e)) s += e;
        checkRep();
    }
    // ...
}
```

The representations used by `CharSet1 / 2 / 3` are not suited for representing sets of arbitrary-type elements. The `String` reps, for example, cannot represent a `Set<Integer>` without careful work to define a new rep invariant and abstraction function that handles multi-digit numbers.

Generic interface, generic implementation. We can also implement the generic `Set<E>` interface without picking a type for `E`. In that case, we write our code blind to the actual type that clients will choose for `E`. Java's `HashSet` ([//docs.oracle.com/javase/8/docs/api/?java/util/HashSet.html](https://docs.oracle.com/javase/8/docs/api/?java/util/HashSet.html)) does that for `Set`. Its declaration looks like:

```
public interface Set<E> {
    // ...
```

```
public class HashSet<E> implements Set<E> {
    // ...
```

A generic implementation can only rely on details of the placeholder types that are included in the interface's specification. We'll see in a future reading how `HashSet` relies on methods that every type in Java is required to implement — and only on those methods, because it can't rely on methods declared in any specific type.

Why Interfaces?

Interfaces are used pervasively in real Java code. Not every class is associated with an interface, but there are a few good reasons to bring an interface into the picture.

- **Documentation for both the compiler and for humans**. Not only does an interface help the compiler catch ADT implementation bugs, but it is also much more useful for a human to read than the code for a concrete implementation. Such an implementation intersperses ADT-level types and specs with implementation details.
- **Allowing performance trade-offs**. Different implementations of the ADT can provide methods with very different performance characteristics. Different applications may work better with different choices, but we would like to code these applications in a way that is representation-independent. From a correctness standpoint, it should be possible to drop in any new implementation of a key ADT with simple, localized code changes.
- **Optional methods**. `List` from the Java standard library marks all mutator methods as optional. By building an implementation that does not support these methods, we can provide immutable lists. Some operations are hard to implement with good enough performance on immutable lists, so we want mutable implementations, too. Code that doesn't call mutators can be written to work automatically with either kind of list.
- **Methods with intentionally underdetermined specifications**. An ADT for finite sets could leave unspecified the element order one gets when converting to a list. Some implementations might use slower method implementations that manage to keep the set representation in some sorted order, allowing quick conversion to a sorted list. Other implementations might make many methods faster by not bothering to support conversion to sorted lists.
- **Multiple views of one class**. A Java class may implement multiple methods. For instance, a user interface widget displaying a drop-down list is natural to view as both a widget and a list. The class for this widget could implement both interfaces. In other words, we don't implement an ADT multiple times just because we are choosing different data structures; we may make multiple implementations because many different sorts of objects may also be seen as special cases of the ADT, among other useful perspectives.
- **More and less trustworthy implementations**. Another reason to implement an interface multiple times might be that it is easy to build a simple implementation that you believe is correct, while you can work harder to build a fancier version that is more likely to contain bugs. You can choose implementations for applications based on how bad it would be to get bitten by a bug.

Realizing ADT Concepts in Java

We can now extend our Java toolbox of ADT concepts ([./12-abstract-data-types/#realizing_adt_concepts_in_java](#)) from the first ADTs reading:

ADT concept	Ways to do it in Java	Examples
Abstract data type	Single class	<code>String</code> (//docs.oracle.com/javase/8/docs/api/java/lang/String.html)
	Interface + class(es)	<code>List</code> (//docs.oracle.com/javase/8/docs/api/java/util/List.html) and <code>ArrayList</code> (//docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)
Creator operation	Constructor	<code>ArrayList()</code> (//docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#ArrayList--)
	Static (factory) method	<code>Collections.singletonList()</code> (//docs.oracle.com/javase/8/docs/api/java/util/Collections.html#singletonList-T-) , <code>Arrays.toList()</code> (//docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList-T...-)
	Constant	<code>BigInteger.ZERO</code> (//docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html#ZERO)
Observer operation	Instance method	<code>List.get()</code> (//docs.oracle.com/javase/8/docs/api/java/util/List.html#get-int-)
	Static method	<code>Collections.max()</code> (//docs.oracle.com/javase/8/docs/api/java/util/Collections.html#max-java.util.Collection-)

Producer operation	Instance method	<code>String.trim()</code> (//docs.oracle.com/javase/8/docs/api/java/lang/String.html#trim--)
	Static method	<code>Collections.unmodifiableList()</code> (//docs.oracle.com/javase/8/docs/api/java/util/Collections.html#unmodifiableList--java.util.List)
Mutator operation	Instance method	<code>List.add()</code> (//docs.oracle.com/javase/8/docs/api/java/util/List.html#add-E)
	Static method	<code>Collections.copy()</code> (//docs.oracle.com/javase/8/docs/api/java/util/Collections.html#copy--java.util.List-java.util.List)
Representation	private fields	

Summary

Java interfaces help us formalize the idea of an abstract data type as a set of operations that must be supported by a type.

This helps make our code...

- **Safe from bugs.** An ADT is defined by its operations, and interfaces do just that. When clients use an interface type, static checking ensures that they only use methods defined by the interface. If the implementation class exposes other methods — or worse, has visible representation — the client can't accidentally see or depend on them. When we have multiple implementations of a data type, interfaces provide static checking of the method signatures.
- **Easy to understand.** Clients and maintainers know exactly where to look for the specification of the ADT. Since the interface doesn't contain instance fields or implementations of instance methods, it's easier to keep details of the implementation out of the specifications.
- **Ready for change.** We can easily add new implementations of a type by adding classes that implement interface. If we avoid constructors in favor of static factory methods, clients will only see the interface. That means we can switch which implementation class clients are using without changing their code at all.

Reading 15: Equality**Introduction****Three Ways to Regard Equality****`==` vs. `equals()`****Equality of Immutable Types****The Object Contract****Equality of Mutable Types****The Final Rule for `equals()` and `hashCode()`****Summary**

Reading 15: Equality

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- Understand equality defined in terms of the abstraction function, an equivalence relation, and observations.
- Differentiate between reference equality and object equality.
- Differentiate between strict observational and behavioral equality for mutable types.
- Understand the Object contract and be able to implement equality correctly for mutable and immutable types.

Introduction

In the previous readings we've developed a rigorous notion of *data abstraction* by creating types that are characterized by their operations, not by their representation. For an abstract data type, the *abstraction function* explains how to interpret a concrete representation value as a value of the abstract type, and we saw how the choice of abstraction function determines how to write the code implementing each of the ADT's operations.

In this reading we turn to how we define the notion of *equality* of values in a data type: the abstraction function will give us a way to cleanly define the equality operation on an ADT.

In the physical world, every object is distinct – at some level, even two snowflakes are different, even if the distinction is just the position they occupy in space. (This isn't strictly true of all subatomic particles, but true enough of large objects like snowflakes and baseballs and people.) So two physical objects are never truly "equal" to each other; they only have degrees of similarity.

In the world of human language, however, and in the world of mathematical concepts, you can have multiple names for the same thing. So it's natural to ask when two expressions represent the same thing: $1+2$, $\sqrt{9}$, and 3 are alternative expressions for the same ideal mathematical value.

Three Ways to Regard Equality

Formally, we can regard equality in several ways.

Using an abstraction function. Recall that an abstraction function $f: R \rightarrow A$ maps concrete instances of a data type to their corresponding abstract values. To use f as a definition for equality, we would say that a equals b if and only if $f(a)=f(b)$.

Using a relation. An *equivalence* is a relation $E \subseteq T \times T$ that is:

- reflexive: $E(t,t) \forall t \in T$
- symmetric: $E(t,u) \Rightarrow E(u,t)$
- transitive: $E(t,u) \wedge E(u,v) \Rightarrow E(t,v)$

To use E as a definition for equality, we would say that a equals b if and only if $E(a,b)$.

These two notions are equivalent. An equivalence relation induces an abstraction function (the relation partitions T , so f maps each element to its partition class). The relation induced by an abstraction function is an equivalence relation (check for yourself that the three properties hold).

A third way we can talk about the equality between abstract values is in terms of what an outsider (a client) can observe about them:

Using observation. We can say that two objects are equal when they cannot be distinguished by observation – every operation we can apply produces the same result for both objects. Consider the set expressions $\{1,2\}$ and $\{2,1\}$. Using the observer operations available for sets, cardinality $|...|$ and membership \in , these expressions are indistinguishable:

- $|\{1,2\}| = 2$ and $|\{2,1\}| = 2$
- $1 \in \{1,2\}$ is true, and $1 \in \{2,1\}$ is true
- $2 \in \{1,2\}$ is true, and $2 \in \{2,1\}$ is true
- $3 \in \{1,2\}$ is false, and $3 \in \{2,1\}$ is false
- ... and so on

In terms of abstract data types, "observation" means calling operations on the objects. So two objects are equal if and only if they cannot be distinguished by calling any operations of the abstract data type.

Example: Duration

Here's a simple example of an immutable ADT.

```
public class Duration {
    private final int mins;
    private final int secs;
    // rep invariant:
    //   mins >= 0, secs >= 0
    // abstraction function:
    //   represents a span of time of mins minutes and secs seconds

    /** Make a duration lasting for m minutes and s seconds. */
    public Duration(int m, int s) {
        mins = m; secs = s;
    }
    /** @return length of this duration in seconds */
    public long getLength() {
        return mins*60 + secs;
    }
}
```

Now which of the following values should be considered equal?

```
Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 3);
Duration d3 = new Duration (0, 62);
Duration d4 = new Duration (1, 2);
```

Think in terms of both the abstraction-function definition of equality, and the observational equality definition.

`==` vs. `equals()`

Like many languages, Java has two different operations for testing equality, with different semantics.

- The `==` operator compares references. More precisely, it tests *referential equality*. Two references are `==` if they point to the same storage in memory. In terms of the snapshot diagrams we've been drawing, two references are `==` if their arrows point to the same object bubble.
- The `equals()` operation compares object contents – in other words, *object equality*, in the sense that we've been talking about in this reading. The `equals()` operation has to be defined appropriately for every abstract data type.

For comparison, here are the equality operators in several languages:

	referential equality	object equality
Java	<code>==</code>	<code>equals()</code>
Objective C	<code>==</code>	<code>isEqual:</code>
C#	<code>==</code>	<code>Equals()</code>
Python	<code>is</code>	<code>==</code>
Javascript	<code>==</code>	n/a

Note that `==` unfortunately flips its meaning between Java and Python. Don't let that confuse you: `==` in Java just tests reference identity, it doesn't compare object contents.

As programmers in any of these languages, we can't change the meaning of the referential equality operator. In Java, `==` always means referential equality. But when we define a new data type, it's our responsibility to decide what object equality means for values of the data type, and implement the `equals()` operation appropriately.

Equality of Immutable Types

The `equals()` method is defined by `Object`, and its default implementation looks like this:

```
public class Object {
    ...
    public boolean equals(Object that) {
        return this == that;
    }
}
```

In other words, the default meaning of `equals()` is the same as referential equality. For immutable data types, this is almost always wrong. So you have to **override** the `equals()` method, replacing it with your own implementation.

Here's our first try for `Duration`:

```
public class Duration {
    ...
    // Problematic definition of equals()
    public boolean equals(Duration that) {
        return this.getLength() == that.getLength();
    }
}
```

There's a subtle problem here. Why doesn't this work? Let's try this code:

```
Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 2);
Object o2 = d2;
d1.equals(d2) → true
d1.equals(o2) → false
```

You can see this code in action

```
(/www.pythontutor.com/java.html#code=public+class+Duration+%7B%0A++++private+final+int+mins%3B%0A++++private+final+int+secs%3B%0A++++//+rep+invar
(1,+2%29%3B%0A+++++Duration+d2+%3D+new+Duration+
(1,+2%29%3B%0A+++++Object+o2+%3D+d2%3B%0A+++++System.out.println(%22d1.equals(d2)%29%3D%22+%2B+d1.equals(d2)%29%29%3B%0A+++++S)
frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=33). You'll see that even though d2
and o2 end up referring to the very same object in memory, you still get different results for them from equals() .
```

What's going on? It turns out that `Duration` has **overloaded** the `equals()` method, because the method signature was not identical to `Object`'s. We actually have two `equals()` methods in `Duration`: an implicit `equals(Object)` inherited from `Object`, and the new `equals(Duration)`.

```
public class Duration extends Object {
    // explicit method that we declared:
    public boolean equals (Duration that) {
        return this.getLength() == that.getLength();
    }
    // implicit method inherited from Object:
    public boolean equals (Object that) {
        return this == that;
    }
}
```

We've seen overloading since the very beginning of the course in static checking ([./01-static-checking/#types](#)). Recall from the Java Tutorials ([//docs.oracle.com/javase/tutorial/java/javaOO/methods.html](#)) that the compiler selects between overloaded operations using the compile-time type of the parameters. For example, when you use the `/` operator, the compiler chooses either integer division or float division based on whether the arguments are ints or floats. The same compile-time selection happens here. If we pass an `Object` reference, as in `d1.equals(o2)`, we end up calling the `equals(Object)` implementation. If we pass a `Duration` reference, as in `d1.equals(d2)`, we end up calling the `equals(Duration)` version. This happens even though `o2` and `d2` both point to the same object at runtime! Equality has become inconsistent.

It's easy to make a mistake in the method signature, and overload a method when you meant to override it. This is such a common error that Java has a language feature, the annotation `@Override` ([https://docs.oracle.com/javase/tutorial/java/annotations/predefined.html](#)), which you should use whenever your intention is to override a method in your superclass. With this annotation, the Java compiler will check that a method with the same signature actually exists in the superclass, and give you a compiler error if you've made a mistake in the signature.

So here's the right way to implement `Duration`'s `equals()` method:

```
@Override
public boolean equals (Object thatObject) {
    if (!(thatObject instanceof Duration)) return false;
    Duration thatDuration = (Duration) thatObject;
    return this.getLength() == thatDuration.getLength();
}
```

This fixes the problem:

```
Duration d1 = new Duration(1, 2);
Duration d2 = new Duration(1, 2);
Object o2 = d2;
d1.equals(d2) → true
d1.equals(o2) → true
```

You can see this code in action

```
(/www.pythontutor.com/java.html#code=public+class+Duration+%7B%0A++++private+final+int+mins%3B%0A++++private+final+int+secs%3B%0A++++//+rep+invar
(Object+thatObject%29+%7B%0A++++++if+!((thatObject+instanceof+Duration)%29%29+return+false%3B%0A++++++Duration+thatDuration%3D+
(Duration%29+thatObject%3B%0A++++++return+this.getLength()%29+%3D%3D+thatDuration.getLength()%29%3B%0A++++%7D%0A++++public+static+void+m=
(1,+2%29%3B%0A+++++Duration+d2+%3D+new+Duration+
(1,+2%29%3B%0A+++++Object+o2+%3D+d2%3B%0A+++++System.out.println(%22d1.equals(d2)%29%3D%22+%2B+d1.equals(d2)%29%29%3B%0A+++++S)
frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=49) in the Online Python Tutor.
```

instanceof

The `instanceof` operator ([https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2.html](#)) tests whether an object is an instance of a particular type. Using `instanceof` is dynamic type checking, not the static type checking we vastly prefer. In general, using `instanceof` in object-oriented programming is a bad smell. In 6.005 — and this is another of our rules that holds true in most good Java programming — **instanceof is disallowed anywhere except for implementing equals**. This prohibition also includes other ways of inspecting objects' runtime types. For example, `getClass` ([//docs.oracle.com/javase/8/docs/api/java/lang/Object.html#getClass--](#)) is also disallowed.

We'll see examples of when you might be tempted to use `instanceof`, and how to write alternatives that are safer from bugs and more ready for change, in a future reading.

The Object Contract

The specification of the `Object` class is so important that it is often referred to as *the Object Contract*. The contract can be found in the method specifications for the `Object` class. Here we will focus on the contract for `equals`. When you override the `equals` method, you must adhere to its general contract. It states that:

- `equals` must define an equivalence relation – that is, a relation that is reflexive, symmetric, and transitive;
- `equals` must be consistent: repeated calls to the method must yield the same result provided no information used in `equals` comparisons on the object is modified;
- for a non-null reference `x`, `x.equals(null)` should return false;
- `hashCode` must produce the same result for two objects that are deemed equal by the `equals` method.

Breaking the Equivalence Relation

Let's start with the equivalence relation. We have to make sure that the definition of equality implemented by `equals()` is actually an equivalence relation as defined earlier: reflexive, symmetric, and transitive. If it isn't, then operations that depend on equality (like sets, searching) will behave erratically and unpredictably. You don't want to program with a data type in which sometimes `a.equals(b)`, but `b` doesn't equal `a`. Subtle and painful bugs will result.

Here's an example of how an innocent attempt to make equality more flexible can go wrong. Suppose we wanted to allow for a tolerance in comparing `Duration` objects, because different computers may have slightly unsynchronized clocks:

```
private static final int CLOCK_SKEW = 5; // seconds

@Override
public boolean equals (Object thatObject) {
    if (!(thatObject instanceof Duration)) return false;
    Duration thatDuration = (Duration) thatObject;
    return Math.abs(this.getLength() - thatDuration.getLength()) <= CLOCK_SKEW;
}
```

Which property of the equivalence relation is violated?

Breaking Hash Tables

To understand the part of the contract relating to the `hashCode` method, you'll need to have some idea of how hash tables work. Two very common collection implementations, `HashSet` and `HashMap`, use a hash table data structure, and depend on the `hashCode` method to be implemented correctly for the objects stored in the set and used as keys in the map.

A hash table is a representation for a mapping: an abstract data type that maps keys to values. Hash tables offer constant time lookup, so they tend to perform better than trees or lists. Keys don't have to be ordered, or have any particular property, except for offering `equals` and `hashCode`.

Here's how a hash table works. It contains an array that is initialized to a size corresponding to the number of elements that we expect to be inserted. When a key and a value are presented for insertion, we compute the hashcode of the key, and convert it into an index in the array's range (e.g., by a modulo division). The value is then inserted at that index.

The rep invariant of a hash table includes the fundamental constraint that keys are in the slots determined by their hash codes.

Hashcodes are designed so that the keys will be spread evenly over the indices. But occasionally a conflict occurs, and two keys are placed at the same index. So rather than holding a single value at an index, a hash table actually holds a list of key/value pairs, usually called a *hash bucket*. A key/value pair is implemented in Java simply as an object with two fields. On insertion, you add a pair to the list in the array slot determined by the hash code. For lookup, you hash the key, find the right slot, and then examine each of the pairs until one is found whose key equals the query key.

Now it should be clear why the `Object` contract requires equal objects to have the same hashcode. If two equal objects had distinct hashcodes, they might be placed in different slots. So if you attempt to lookup a value using a key equal to the one with which it was inserted, the lookup may fail.

`Object`'s default `hashCode()` implementation is consistent with its default `equals()`:

```
public class Object {
    ...
    public boolean equals(Object that) { return this == that; }
    public int hashCode() { return /* the memory address of this */; }
}
```

For references `a` and `b`, if `a == b`, then the address of `a ==` the address of `b`. So the `Object` contract is satisfied.

But immutable objects need a different implementation of `hashCode()`. For `Duration`, since we haven't overridden the default `hashCode()` yet, we're currently breaking the `Object` contract:

```
Duration d1 = new Duration(1, 2);
Duration d2 = new Duration(1, 2);
d1.equals(d2) → true
d1.hashCode() → 2392
d2.hashCode() → 4823
```

`d1` and `d2` are `equal()`, but they have different hash codes. So we need to fix that.

A simple and drastic way to ensure that the contract is met is for `hashCode` to always return some constant value, so every object's hash code is the same. This satisfies the `Object` contract, but it would have a disastrous performance effect, since every key will be stored in the same slot, and every lookup will degenerate to a linear search along a long list.

The standard way to construct a more reasonable hash code that still satisfies the contract is to compute a hash code for each component of the object that is used in the determination of equality (usually by calling the `hashCode` method of each component), and then combining these, throwing in a few arithmetic operations. For `Duration`, this is easy, because the abstract value of the class is already an integer value:

```
@Override
public int hashCode() {
    return (int) getLength();
```

Josh Bloch's fantastic book, *Effective Java*, explains this issue in more detail, and gives some strategies for writing decent hash code functions. The advice is summarized in a good StackOverflow post ([//stackoverflow.com/questions/113511/hash-code-implementation](https://stackoverflow.com/questions/113511/hash-code-implementation)). Recent versions of Java now have a utility method `Objects.hash()` ([//docs.oracle.com/javase/8/docs/api/java/util/Objects.html#hash-Java.lang.Object...](https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html#hash-Java.lang.Object...)) that makes it easier to implement a hash code involving multiple fields.

Note, however, that as long as you satisfy the requirement that equal objects have the same hash code value, then the particular hashing technique you use doesn't make a difference to the correctness of your code. It may affect its performance, by creating unnecessary collisions between different objects, but even a poorly-performing hash function is better than one that breaks the contract.

Most crucially, note that if you don't override `hashCode` at all, you'll get the one from `Object`, which is based on the address of the object. If you have overridden `equals`, this will mean that you will have almost certainly violated the contract. So as a general rule:

Always override `hashCode` when you override `equals`.

Many years ago in (a precursor to 6.005 confusingly numbered) 6.170, a student spent hours tracking down a bug in a project that amounted to nothing more than misspelling `hashCode` as `hashcode`. This created a new method that didn't override the `hashCode` method of `Object` at all, and strange things happened. Use `@Override`!

Equality of Mutable Types

We've been focusing on equality of immutable objects so far in this reading. What about mutable objects?

Recall our definition: two objects are equal when they cannot be distinguished by observation. With mutable objects, there are two ways to interpret this:

- when they cannot be distinguished by observation *that doesn't change the state of the objects*, i.e., by calling only observer, producer, and creator methods. This is often strictly called **observational equality**, since it tests whether the two objects "look" the same, in the current state of the program.
- when they cannot be distinguished by *any* observation, even state changes. This interpretation allows calling any methods on the two objects, including mutators. This is often called **behavioral equality**, since it tests whether the two objects will "behave" the same, in this and all future states.

For immutable objects, observational and behavioral equality are identical, because there aren't any mutator methods.

For mutable objects, it's tempting to implement strict observational equality. Java uses observational equality for most of its mutable data types, in fact. If two distinct `List` objects contain the same sequence of elements, then `equals()` reports that they are equal.

But using observational equality leads to subtle bugs, and in fact allows us to easily break the rep invariants of other collection data structures. Suppose we make a `List`, and then drop it into a `Set`:

```
List<String> list = new ArrayList<>();
list.add("a");

Set<List<String>> set = new HashSet<List<String>>();
set.add(list);
```

We can check that the set contains the list we put in it, and it does:

```
set.contains(list) → true
```

But now we mutate the list:

```
list.add("goodbye");
```

And it no longer appears in the set!

```
set.contains(list) → false!
```

It's worse than that, in fact: when we iterate over the members of the set, we still find the list in there, but `contains()` says it's not there!

```
for (List<String> l : set) {
    set.contains(l) → false!
}
```

If the set's own iterator and its own `contains()` method disagree about whether an element is in the set, then the set clearly is broken. You can see this code in action

([What's going on? `List<String>` is a mutable object. In the standard Java implementation of collection classes like `List`, mutations affect the result of `equals\(\)` and `hashCode\(\)`. When the list is first put into the `HashSet`, it is stored in the hash bucket corresponding to its `hashCode\(\)` result at that time. When the list is subsequently mutated, its `hashCode\(\)` changes, but `HashSet` doesn't realize it should be moved to a different bucket. So it can never be found again.](https://www.pythontutor.com/java.html#code=import+java.util.*%3B%0Apublic+class+WhyObservationalEqualityHurts+%7B%0A++public+static+void+main(String%5B%5E%29%3CString%3E+l+%3A+set%29+%7B%0A++++System.out.println(%22set.contains(%29%3D%22+%2B+set.contains(%29%29%3B%0A++++%7D%0A++frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=13) on Online Python Tutor.</p>
</div>
<div data-bbox=)

When `equals()` and `hashCode()` can be affected by mutation, we can break the rep invariant of a hash table that uses that object as a key.

Here's a telling quote from the specification of `java.util.Set`:

Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set.

The Java library is unfortunately inconsistent about its interpretation of `equals()` for mutable classes. Collections use observational equality, but other mutable classes (like `StringBuilder`) use behavioral equality.

The lesson we should draw from this example is that `equals()` should implement behavioral equality. In general, that means that two references should be `equals()` if and only if they are aliases for the same object. So mutable objects should just inherit `equals()` and `hashCode()` from `Object`. For clients that need a notion of observational equality (whether two mutable objects “look” the same in the current state), it’s better to define a new method, e.g., `similar()`.

The Final Rule for `equals()` and `hashCode()`

For immutable types :

- `equals()` should compare abstract values. This is the same as saying `equals()` should provide behavioral equality.
- `hashCode()` should map the abstract value to an integer.

So immutable types must override both `equals()` and `hashCode()`.

For mutable types :

- `equals()` should compare references, just like `==`. Again, this is the same as saying `equals()` should provide behavioral equality.
- `hashCode()` should map the reference into an integer.

So mutable types should not override `equals()` and `hashCode()` at all, and should simply use the default implementations provided by `Object`. Java doesn’t follow this rule for its collections, unfortunately, leading to the pitfalls that we saw above.

Autoboxing and Equality

One more instructive pitfall in Java. We’ve talked about primitive types and their object type equivalents – for example, `int` and `Integer`. The object type implements `equals()` in the correct way, so that if you create two `Integer` objects with the same value, they’ll be `equals()` to each other:

```
Integer x = new Integer(3);
Integer y = new Integer(3);
x.equals(y) → true
```

But there’s a subtle problem here; `==` is overloaded. For reference types like `Integer`, it implements referential equality:

```
x == y // returns false
```

But for primitive types like `int`, `==` implements behavioral equality:

```
(int)x == (int)y // returns true
```

So you can’t really use `Integer` interchangeably with `int`. The fact that Java automatically converts between `int` and `Integer` (this is called *autoboxing* and *autounboxing*) can lead to subtle bugs! You have to be aware what the compile-time types of your expressions are. Consider this:

```
Map<String, Integer> a = new HashMap(), b = new HashMap();
a.put("c", 130); // put ints into the map
b.put("c", 130);
a.get("c") == b.get("c") → ?? // what do we get out of the map?
```

You can see this code in action

([https://www.pythontutor.com/java.html#code=import+java.util.*%3B%0Apublic+class+AutounboxingProblem+%7B%0A++public+static+void+main\(String%5B%5D+args%0A.get\(%22c%22%29+%3D%3D+b.get\(%22c%22%29%29+%29%3B%0A++%7D%0A%7D&mode=display&origin=opt-frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=6](https://www.pythontutor.com/java.html#code=import+java.util.*%3B%0Apublic+class+AutounboxingProblem+%7B%0A++public+static+void+main(String%5B%5D+args%0A.get(%22c%22%29+%3D%3D+b.get(%22c%22%29%29+%29%3B%0A++%7D%0A%7D&mode=display&origin=opt-frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=6)) on Online Python Tutor.

Summary

- Equality should be an equivalence relation (reflexive, symmetric, transitive).
- Equality and hash code must be consistent with each other, so that data structures that use hash tables (like `HashSet` and `HashMap`) work properly.
- The abstraction function is the basis for equality in immutable data types.
- Reference equality is the basis for equality in mutable data types; this is the only way to ensure consistency over time and avoid breaking rep invariants of hash tables.

Equality is one part of implementing an abstract data type, and we’ve already seen how important ADTs are to achieving our three primary objectives. Let’s look at equality in particular:

- **Safe from bugs**. Correct implementation of equality and hash codes is necessary for use with collection data types like sets and maps. It’s also highly desirable for writing tests. Since every object in Java inherits the `Object` implementations, immutable types must override them.
- **Easy to understand**. Clients and other programmers who read our specs will expect our types to implement an appropriate equality operation, and will be surprised and confused if we do not.
- **Ready for change**. Correctly-implemented equality for *immutable* types separates equality of reference from equality of abstract value, hiding from clients our decisions about whether values are shared. Choosing behavioral rather than observational equality for *mutable* types helps avoid unexpected aliasing bugs.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 16: Recursive Data Types

Introduction

Part 1: Recursive Data Types

Part 2: Writing a Program with ADTs

Summary

Reading 16: Recursive Data Types

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- Understand recursive datatypes
- Read and write datatype definitions
- Understand and implement functions over recursive datatypes
- Understand immutable lists and know the standard operations on immutable lists
- Know and follow a recipe for writing programs with ADTs

Introduction

In this reading we'll look at recursively-defined types, how to specify operations on such types, and how to implement them. Our main example will be *immutable lists*.

Then we'll use another recursive datatype example, *matrix multiplications*, to walk through our process for programming with ADTs.

Part 1: Recursive Data Types (recursive/)

Part 2: Writing a Program with ADTs (matexpr/)

Summary

Let's review how recursive datatypes fit in with the main goals of this course:

- **Safe from bugs**. Recursive datatypes allow us to tackle problems with a recursive or unbounded structure. Implementing appropriate data structures that encapsulate important operations and maintain their own invariants is crucial for correctness.

- **Easy to understand** . Functions over recursive datatypes, specified in the abstract type and implemented in each concrete variant, organize the different behavior of the type.
- **Ready for change** . A recursive ADT, like any ADT, separates abstract values from concrete representations, making it possible to change low-level code and high-level structure of the implementation without changing clients.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 17: Regular Expressions & Grammars

Introduction

Grammars

Regular Expressions

Summary

Reading 17: Regular Expressions & Grammars

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

After today's class, you should:

- Understand the ideas of grammar productions and regular expression operators
- Be able to read a grammar or regular expression and determine whether it matches a sequence of characters
- Be able to write a grammar or regular expression to match a set of character sequences and parse them into a data structure

Introduction

Today's reading introduces several ideas:

- grammars, with productions, nonterminals, terminals, and operators
- regular expressions
- parser generators

Some program modules take input or produce output in the form of a sequence of bytes or a sequence of characters, which is called a *string* when it's simply stored in memory, or a *stream* when it flows into or out of a module. In today's reading, we talk about how to write a specification for such a sequence.

Concretely, a sequence of bytes or characters might be:

- A file on disk, in which case the specification is called the *file format*
- Messages sent over a network, in which case the specification is a *wire protocol*
- A command typed by the user on the console, in which case the specification is a *command line interface*
- A string stored in memory

For these kinds of sequences, we introduce the notion of a *grammar*, which allows us not only to distinguish between legal and illegal sequences, but also to parse a sequence into a data structure that a program can work with. The data structure produced from a grammar will often be a recursive data type like we talked about in the recursive data type reading (./16-recursive-data-types/).

We also talk about a specialized form of a grammar called a *regular expression*. In addition to being used for specification and parsing, regular expressions are a widely-used tool for many string-processing tasks that need to disassemble a string, extract information from it, or transform it.

The next reading will talk about parser generators, a kind of tool that translates a grammar automatically into a parser for that grammar.

Grammars

To describe a sequence of symbols, whether they are bytes, characters, or some other kind of symbol drawn from a fixed set, we use a compact representation called a *grammar*.

A *grammar* defines a set of sentences, where each *sentence* is a sequence of symbols. For example, our grammar for URLs will specify the set of sentences that are legal URLs in the HTTP protocol.

The symbols in a sentence are called *terminals* (or tokens).

They're called terminals because they are the leaves of a tree that represents the structure of the sentence. They don't have any children, and can't be expanded any further. We generally write terminals in quotes, like 'http' or ':' .

A grammar is described by a set of *productions*, where each production defines a *nonterminal*. You can think of a nonterminal like a variable that stands for a set of sentences, and the production as the definition of that variable in terms of other variables (nonterminals), operators, and constants (terminals). Nonterminals are internal nodes of the tree representing a sentence.

A production in a grammar has the form

nonterminal ::= expression of terminals, nonterminals, and operators

In 6.005, we will name nonterminals using lowercase identifiers, like `x` or `y` or `url` .

One of the nonterminals of the grammar is designated as the *root*. The set of sentences that the grammar recognizes are the ones that match the root nonterminal. This nonterminal is often called `root` or `start` , but in the grammars below we will typically choose more memorable names like `url` , `html` , and `markdown` .

Grammar Operators

The three most important operators in a production expression are:

- concatenation

`x ::= y z` an `x` is a `y` followed by a `z`

- repetition

`x ::= y*` an `x` is zero or more `y`

- union (also called alternation)

`x ::= y | z` an `x` is a `y` or a `z`

You can also use additional operators which are just syntactic sugar (i.e., they're equivalent to combinations of the big three operators):

- option (0 or 1 occurrence)

```
x ::= y?      an x is a y or is the empty sentence
```

- 1+ repetition (1 or more occurrences)

```
x ::= y+      an x is one or more y
               (equivalent to x ::= y y*)
```

- character classes

```
x ::= [abc]  is equivalent to x ::= 'a' | 'b' | 'c'
```

```
x ::= [^b]    is equivalent to x ::= 'a' | 'c' | 'd' | 'e' | 'f'
               | ... (all other characters)
```

By convention, the operators `*`, `?`, and `+` have highest precedence, which means they are applied first. Alternation `|` has lowest precedence, which means it is applied last. Parentheses can be used to override this precedence, so that a sequence or alternation can be repeated:

- grouping using parentheses

```
x ::= (y z | a b)*  an x is zero or more y-z or a-b pairs
```

Example: URL

Suppose we want to write a grammar that represents URLs. Let's build up a grammar gradually by starting with simple examples and extending the grammar as we go.

Here's a simple URL:

```
http://mit.edu/
```

A grammar that represents the set of sentences containing *only this URL* would look like:

```
url ::= 'http://mit.edu/'
```

But let's generalize it to capture other domains, as well:

```
http://stanford.edu/
http://google.com/
```

We can write this as one line, like this:

```
url ::= 'http://' [a-z]+ '.' [a-z]+ '/'
```

url

|

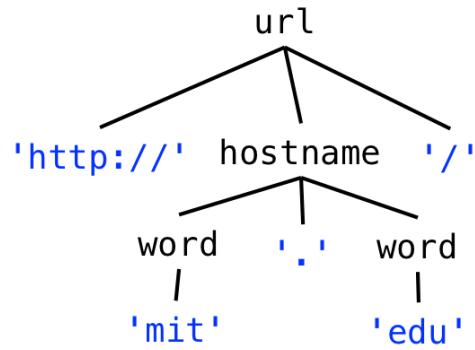
'http://mit.edu/'

This grammar represents the set of all URLs that consist of just a two-part hostname, where each part of the hostname consists of 1 or more letters. So `http://mit.edu/` and `http://yahoo.com/` would match, but not `http://ou812.com/`. Since it has only one nonterminal, a *parse tree* for this URL grammar would look like the picture on the right.

In this one-line form, with a single nonterminal whose production uses only operators and terminals, a grammar is called a *regular expression* (more about that later). But it will be easier to understand if we name the parts using new nonterminals:

```
url ::= 'http://' hostname '/'
hostname ::= word '.' word
word ::= [a-z]+
```

The parse tree for this grammar is now shown at right. The tree has more structure now. The leaves of the tree are the parts of the string that have been parsed. If we concatenated the leaves together, we would recover the original string. The `hostname` and `word` nonterminals are labeling nodes of the tree whose subtrees match those rules in the grammar. Notice that the immediate children of a nonterminal node like `hostname` follow the pattern of the `hostname` rule, `word '.' word`.



How else do we need to generalize? Hostnames can have more than two components, and there can be an optional port number:

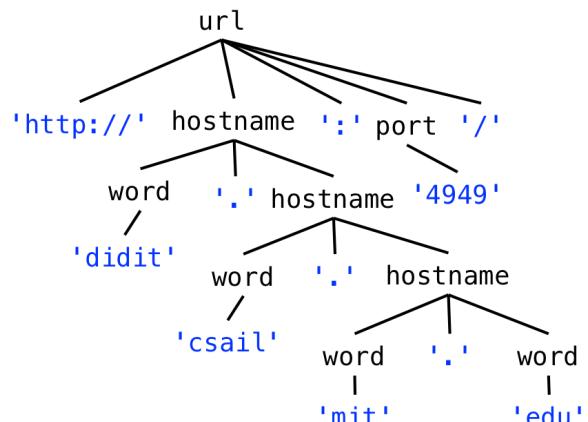
```
http://diddit.csail.mit.edu:4949/
```

To handle this kind of string, the grammar is now:

```
url ::= 'http://' hostname (':' port)?
      '/'

hostname ::= word '.' hostname | word '.'
word
port ::= [0-9]+
word ::= [a-z]+
```

Notice how `hostname` is now defined recursively in terms of itself. Which part of the `hostname` definition is the base case, and which part is the recursive step? What kinds of hostnames are allowed?



Using the repetition operator, we could also write `hostname` like this:

```
hostname ::= (word '.')+ word
```

Another thing to observe is that this grammar allows port numbers that are not technically legal, since port numbers can only range from 0 to 65535. We could write a more complex definition of `port` that would allow only these integers, but that's not typically done in a grammar. Instead, the constraint `0 <= port <= 65535` would be specified alongside the grammar.

There are more things we should do to go farther:

- generalizing `http` to support the additional protocols that URLs can have
- generalizing the `/` at the end to a slash-separated path
- allowing hostnames with the full set of legal characters instead of just `a-z`

Example: Markdown and HTML

Now let's look at grammars for some file formats. We'll be using two different markup languages that represent typographic style in text. Here they are:

Markdown

This is italic.

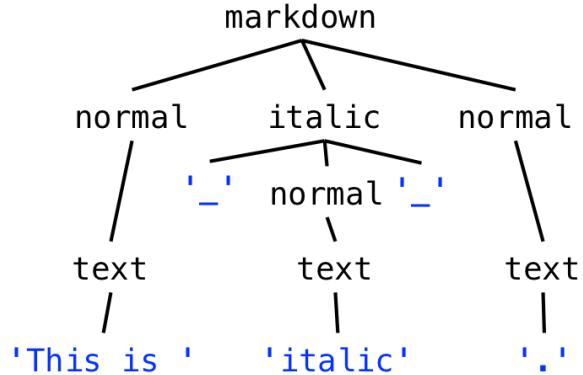
HTML

Here is an *italic* word.

For simplicity, our example HTML and Markdown grammars will only specify italics, but other text styles are of course possible.

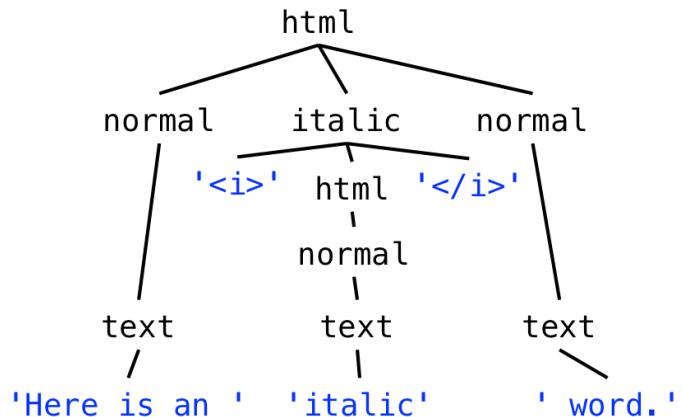
Here's the grammar for our simplified version of Markdown:

```
markdown ::= ( normal | italic ) *
italic ::= '_' normal '_'
normal ::= text
text ::= [^_]*
```



Here's the grammar for our simplified version of HTML:

```
html ::= ( normal | italic ) *
italic ::= '<i>' html '</i>'
normal ::= text
text ::= [^<>]*
```



Regular Expressions

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.

Our URL grammar was regular. By replacing nonterminals with their productions, it can be reduced to a single expression:

```
url ::= 'http://' ([a-z]+ '.')+ [a-z]+ (':' [0-9]+)? '/'
```

The Markdown grammar is also regular:

```
markdown ::= ([^_]*)|(''[^_]*'')*
```

But our HTML grammar can't be reduced completely. By substituting righthand sides for nonterminals, you can eventually reduce it to something like this:

```
html ::= ( [^<>]* | '<i>' html '</i>' ) *
```

...but the recursive use of `html` on the righthand side can't be eliminated, and can't be simply replaced by a repetition operator either. So the HTML grammar is not regular.

The reduced expression of terminals and operators can be written in an even more compact form, called a *regular expression*. A regular expression does away with the quotes around the terminals, and the spaces between terminals and operators, so that it consists just of terminal characters, parentheses for grouping, and operator characters. For example, the regular expression for our `markdown` format is just

```
([^_*]*|_*[^_*]*_)*
```

Regular expressions are also called *regexes* for short. A regex is far less readable than the original grammar, because it lacks the nonterminal names that documented the meaning of each subexpression. But a regex is fast to implement, and there are libraries in many programming languages that support regular expressions.

The regex syntax commonly implemented in programming language libraries has a few more special operators, in addition to the ones we used above in grammars. Here's are some common useful ones:

- any single character
- `\d` any digit, same as `[0-9]`
- `\s` any whitespace character, including space, tab, newline
- `\w` any word character, including letters and digits
- `\., \(, \)`, `*, \+, ...`
escapes an operator or special character so that it matches literally

Using backslashes is important whenever there are terminal characters that would be confused with special characters. Because our `url` regular expression has `.` in it as a terminal, we need to use a backslash to escape it:

```
http://([a-z]+\.)+[a-z]+(:[0-9]+)/*
```

Using regular expressions in Java

Regular expressions (“regexes”) are widely used in programming, and you should have them in your toolbox.

In Java, you can use regexes for manipulating strings (see `String.split` ([, `String.matches` \(\[, `java.util.regex.Pattern` \\(<https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html>\\)\\). They're built-in as a first-class feature of modern scripting languages like Python, Ruby, and Javascript, and you can use them in many text editors for find and replace. Regular expressions are your friend! Most of the time. Here are some examples.\]\(https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#matches-java.lang.String-\)](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#split-java.lang.String-)

Replace all runs of spaces with a single space:

```
String singleSpacedString = string.replaceAll(" +", " ");
```

Match a URL:

```
Pattern regex = Pattern.compile("http://([a-z]+\.\.)+[a-z]+(:[0-9]+)?/");  
Matcher m = regex.matcher(string);  
if (m.matches()) {  
    // then string is a url  
}
```

Extract part of an HTML tag:

```
Pattern regex = Pattern.compile("<a href=['\"]([^\"]*)['\"]>");  
Matcher m = regex.matcher(string);  
if (m.matches()) {  
    String url = m.group(1);  
    // Matcher.group(n) returns the nth parenthesized part of the regex  
}
```

Notice the backslashes in the URL and HTML tag examples. In the URL example, we want to match a literal period . , so we have to first escape it as \. to protect it from being interpreted as the regex match-any-character operator, and then we have to further escape it as \\. to protect the backslash from being interpreted as a Java string escape character. In the HTML example, we have to escape the quote mark " as \' to keep it from ending the string. The frequency of backslash escapes makes regexes still less readable.

Context-Free Grammars

In general, a language that can be expressed with our system of grammars is called context-free. Not all context-free languages are also regular; that is, some grammars can't be reduced to single nonrecursive productions. Our HTML grammar is context-free but not regular.

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 but not regular. That description applies to the Java grammar, shown here in part:

```
statement ::=  
{' statement* '}'  
| 'if' '(' expression ')' statement ('else' statement)?  
| 'for' '(' forinit? ';' expression? ';' forupdate? ')' statement  
| 'while' '(' expression ')' statement  
| 'do' statement 'while' '(' expression ')' ';' '  
| 'try' '{}' statement* '{}' ( catches | catches? 'finally' '{}' statement* '{}')  
| 'switch' '(' expression ')' '{}' switchgroups '{}'  
| 'synchronized' '(' expression ')' '{}' statement* '{}'  
| 'return' expression? ';' '  
| 'throw' expression ';' '  
| 'break' identifier? ';' '  
| 'continue' identifier? ';' '  
| expression ';' '  
| identifier ':' statement  
| ;'
```

Summary

Machine-processed textual languages are ubiquitous in computer science. Grammars are the most popular formalism for describing such languages, and regular expressions are an important subclass of grammars that can be expressed without recursion.

The topics of today's reading connect to our three properties of good software as follows:

- **Safe from bugs.** Grammars and regular expressions are declarative specifications for strings and streams, which can be used directly by libraries and tools. These specifications are often simpler, more direct, and less likely to be buggy than parsing code written by hand.
- **Easy to understand.** A grammar captures the shape of a sequence in a form that is easier to understand than hand-written parsing code. Regular expressions, alas, are often not easy to understand, because they are a one-line reduced form of what might have been a more understandable regular grammar.
- **Ready for change.** A grammar can be easily edited, but regular expressions, unfortunately, are much harder to change, because a complex regular expression is cryptic and hard to understand.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 18: Parser Generators

Parser Generators

An Antlr Grammar

Generating the parser

Calling the parser

Traversing the parse tree

Constructing an abstract syntax tree

Handling errors

Summary

Reading 18: Parser Generators

Software in 6.005

Main.java line 521 (<https://github.com/mit-6.005/s16/tree/18-parser-generators/blob/master/src/intexpr/Main.java#L521-L201>)

~~Safe from bugs~~

~~Easy to understand~~

Ready for change

Correct today and correct in the unknown future.

Communicating clearly with future programmers, including future you.

Designed to accommodate change without rewriting.

Objectives

After today's class, you should:

- Be able to use a grammar in combination with a parser generator, to parse a character sequence into a parse tree
- Be able to convert a parse tree into a useful data type

Parser Generators

A *parser generator* is a good tool that you should make part of your toolbox. A parser generator takes a grammar as input and automatically generates source code that can parse streams of characters using the grammar.

The generated code is a *parser*, which takes a sequence of characters and tries to match the sequence against the grammar. The parser typically produces a *parse tree*, which shows how grammar productions are expanded into a sentence that matches the character sequence. The root of the parse tree is the starting nonterminal of the grammar. Each node of the parse tree expands into one production of the grammar. We'll see how a parse tree actually looks in the next section.

The final step of parsing is to do something useful with this parse tree. We're going to translate it into a value of a recursive data type. Recursive abstract data types are often used to represent an expression in a language, like HTML, or Markdown, or Java, or algebraic expressions. A recursive abstract data type that represents a language expression is called an *abstract syntax tree* (AST).

Antlr is a mature and widely-used parser generator for Java, and other languages as well. The remainder of this reading will get you started with Antlr. If you run into trouble and need a deeper reference, you can look at:

- Definitive Antlr 4 Reference (<https://www.google.com/search?q=The+Definitive+Antlr+4+Reference>) . A book about Antlr, both tutorial and reference.
- Antlr 4 Documentation Wiki (<https://theantlrguy.atlassian.net/wiki/display/ANTLR4/ANTLR+4+Documentation>) . Concise documentation of the grammar file syntax.
- Antlr 4 Runtime API ([//www.antlr.org/api/Java/](https://www.antlr.org/api/Java/)) . Reference documentation for Antlr's Java classes and interfaces.

An Antlr Grammar

The code for the examples that follow can be found on GitHub as **sp16-ex18-parser-generators** (<https://github.com/mit6005/sp16-ex18-parser-generators>) .

Here is what our HTML grammar looks like as an Antlr source file:

```
grammar Html;

root : html EOF;
html : ( italic | normal ) *;
italic : '<i>' html '</i>';
normal : TEXT;
TEXT : ~[<>]+; /* represents a string of one or more characters that are not < or >
```

Let's break it down.

Each Antlr rule consists of a name, followed by a colon, followed by its definition, terminated by a semicolon.

Nonterminals in Antlr have to be lowercase: `root` , `html` , `normal` , `italic` . Terminals are either quoted strings, like '`<i>`' , or capitalized names, like `EOF` and `TEXT` .

```
root : html EOF;
```

`root` is the entry point of the grammar. This is the nonterminal that the whole input needs to match. We don't have to call it `root` . The entry point can be any nonterminal.

`EOF` is a special terminal, defined by Antlr, that means the end of the input. It stands for *end of file* , though your input may also come from a string or a network connection rather than just a file.

```
html : ( normal | italic ) *;
```

This rule shows that Antlr rules can have the alternation operator `|` , the repetition operators `*` and `+` , and parentheses for grouping, in the same way we've been using in the grammars reading [\(../17-regex-grammars/#grammars\)](#) . Optional parts can be marked with `?` , just like we did earlier, but this particular grammar doesn't use `?` .

```
italic : '<i>' html '</i>';
normal : TEXT;
TEXT : ~[<>]+;
```

`TEXT` is a terminal matching sequences of characters that are neither `<` nor `>`. In the more conventional regular expression syntax used earlier in this reading, we would write `[^<>]` to represent all characters except `<` and `>`. Antlr uses a slightly different syntax – `~` means *not*, and it is put in front of the square brackets instead of inside them, so `~[<>]` matches any character except `<` and `>`.

In Antlr, terminals can be defined using regular expressions, not just fixed strings. For example, here are some other terminal patterns we used in the URL grammar earlier in the reading, now written in Antlr syntax and with Antlr's required naming convention:

```
IDENTIFIER : [a-z]+;
INTEGER : [0-9]+;
```

More about Antlr's grammar file syntax can be found in Chapter 5 of the Definitive ANTLR 4 Reference (<https://www.google.com/search?q=The+Definitive+Antlr+4+Reference>) .

Generating the parser

The rest of this reading will focus on the `IntegerExpression` grammar used in the exercise above, which we'll store in a file called `IntegerExpression.g4` (<https://github.com/mit6005/sp16-ex18-parser-generators/blob/master/src/intexpr/parser/IntegerExpression.g4>). Antlr 4 grammar files end with `.g4` by convention.

The Antlr parser generator tool converts a grammar source file like `IntegerExpression.g4` into Java classes that implement a parser. To do that, you need to go to a command prompt (Terminal or Command Prompt) and run a command like this:

```
cd <root of project>
cd src/intexpr/parser
java -jar ../../lib/antlr.jar IntegerExpression.g4
```

You need to make sure you `cd` into right folder (where `IntegerExpression.g4` is) and you refer to `antlr.jar` where it is in your project folder structure, using a relative path like `../../lib/antlr.jar`).

Assuming you don't have any syntax errors in your grammar file, the parser generator will produce new Java source files in the current folder. Nothing will be printed in the terminal. The generated code is divided into several cooperating modules:

- the **lexer** takes a stream of characters as input, and produces a stream of terminals (Antlr calls them *tokens*) as output, like `NUMBER` , `+` , and `(` . For `IntegerExpression.g4` , the generated lexer is called `IntegerExpressionLexer.java` .
- the **parser** takes the stream of terminals produced by the lexer and produces a parse tree. The generated parser is called `IntegerExpressionParser.java` .
- the **tree walker** lets you write code that walks over the parse tree produced by the parser, as explained below. The generated tree walker files are the interface `IntegerExpressionListener.java` , and an empty implementation of the interface, `IntegerExpressionBaseListener.java` .

Antlr also generates two text files, `IntegerExpression.tokens` and `IntegerExpressionLexer.tokens`, that list the terminals that Antlr found in your grammar. These aren't needed for a simple parser, but they're needed when you include grammars inside other grammars.

Make sure that you:

- **Never edit the files generated by Antlr.** The right way to change your parser is to edit the grammar source file, `IntegerExpression.g4`, and then regenerate the Java classes.
- **Regenerate the files whenever you edit the grammar file.** This is easy to forget when Eclipse is compiling all your Java source files automatically. Eclipse does not regenerate your parser automatically. Make sure you rerun the `java -jar ...` command whenever you change your `.g4` file.
- **Refresh your project in Eclipse each time you regenerate the files.** You can do this by clicking on your project in Eclipse and pressing F5, or right-clicking and choosing Refresh. The reason is that Eclipse sometimes uses older versions of files already part of the source, even if they have been modified on your filesystem.

More about using the Antlr parser generator to produce a parser can be found in Chapter 3 of the Definitive ANTLR 4 Reference (<https://www.google.com/search?q=The+Definitive+Antlr+4+Reference>) .

Calling the parser

Now that you've generated the Java classes for your parser, you'll want to use them from your own code.

First we need to make a stream of characters to feed to the lexer. Antlr has a class `ANTLRInputStream` that makes this easy. It can take a `String`, or a `Reader`, or an `InputStream` as input. Here we are using a string:

```
CharStream stream = new ANTLRInputStream("54+(2+89)");
```

Next, we create an instance of the lexer class that our grammar file generated, and pass it the character stream:

```
IntegerExpressionLexer lexer = new IntegerExpressionLexer(stream);
TokenStream tokens = new CommonTokenStream(lexer);
```

The result is a stream of terminals, which we can then feed to the parser:

```
IntegerExpressionParser parser = new IntegerExpressionParser(tokens);
```

To actually do the parsing, we call a particular nonterminal on the parser. The generated parser has one method for every nonterminal in our grammar, including `root()`, `sum()`, and `primitive()`. We want to call the nonterminal that represents the set of strings that we want to match – in this case, `root()`.

Calling it produces a parse tree:

```
ParseTree tree = parser.root();
```

For debugging, we can then print this tree out:

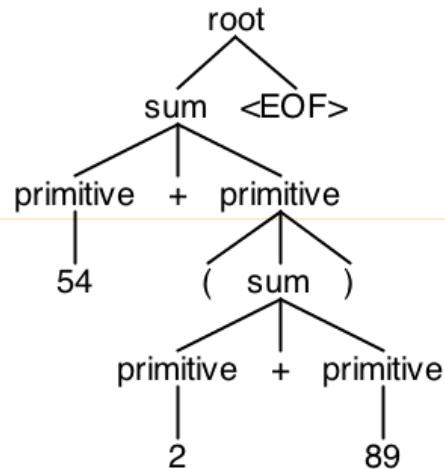
```
System.out.println(tree.toStringTree(parser));
```

Or we can display it in a handy graphical form:

```
Trees.inspect(tree, parser);
```

which pops up a window with the parse tree shown on the right.

In the example code: `Main.java` lines 34-50 (<https://github.com/mit6005/sp16-ex18-parser-generators/blob/master/src/intexpr/Main.java#L34-L50>) , which use lines 71-85 (<https://github.com/mit6005/sp16-ex18-parser-generators/blob/master/src/intexpr/Main.java#L71-L85>) .



Traversing the parse tree

So we've used the parser to turn a stream of characters into a parse tree, which shows how the grammar matches the stream. Now we need to do something with this parse tree. We're going to translate it into a value of a recursive abstract data type.

The first step is to learn how to traverse the parse tree. To do this, we use a `ParseTreeWalker` , which is an Antlr class that walks over a parse tree, visiting every node in order, top-to-bottom, left-to-right. As it visits each node in the tree, the walker calls methods on a *listener* object that we provide, which implements `IntegerExpressionListener` interface.

Just to warm up, here's a simple implementation of `IntegerExpressionListener` that just prints a message every time the walker calls us, so we can see how it gets used:

```

class PrintEverything implements IntegerExpressionListener {

    @Override public void enterRoot(IntegerExpressionParser.RootContext context)
    {
        System.out.println("entering root");
    }
    @Override public void exitRoot(IntegerExpressionParser.RootContext context) {
        System.out.println("exiting root");
    }

    @Override public void enterSum(IntegerExpressionParser.SumContext context) {
        System.out.println("entering sum");
    }
    @Override public void exitSum(IntegerExpressionParser.SumContext context) {
        System.out.println("exiting sum");
    }

    @Override public void enterPrimitive(IntegerExpressionParser.PrimitiveContext
context) {
        System.out.println("entering primitive");
    }
    @Override public void exitPrimitive(IntegerExpressionParser.PrimitiveContext
context) {
        System.out.println("exiting primitive");
    }

    @Override public void visitTerminal(TerminalNode terminal) {
        System.out.println("terminal " + terminal.getText());
    }

    // don't need these here, so just make them empty implementations
    @Override public void enterEveryRule(ParserRuleContext context) { }
    @Override public void exitEveryRule(ParserRuleContext context) { }
    @Override public void visitErrorNode(ErrorNode node) { }
}

```

Notice that every nonterminal N in the grammar has corresponding `enter N ()` and `exit N ()` methods in the listener interface, which are called when the tree walk enters and exits a parse tree node for nonterminal N , respectively. There is also a `visitTerminal()` that is called when the walk reaches a leaf of the parse tree. Each of these methods has a parameter that provides information about the nonterminal or terminal node that the walk is currently visiting.

The listener interface also has some methods that we don't need. The methods `enterEveryRule()` and `exitEveryRule()` are called on entering and exiting *any* nonterminal node, in case we want some generic behavior. The method `visitErrorNode()` is called if the input contained a syntax error that produced an error node in the parse tree. In the parser we're writing, however, a syntax error causes an exception to be thrown, so we won't see any parse trees with error nodes in them. The interface requires us to implement these methods, but we can just leave their method bodies empty.

```

ParseTreeWalker walker = new ParseTreeWalker();
IntegerExpressionListener listener = new PrintEverything();
walker.walk(listener, tree);

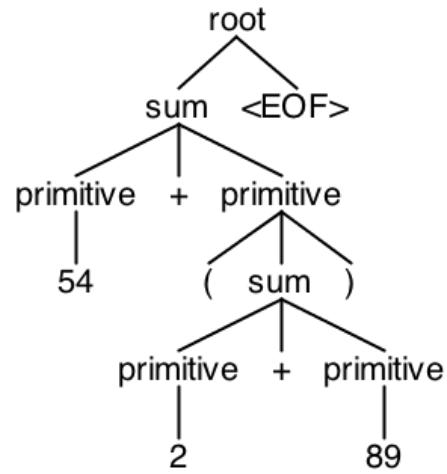
```

If we walk over the parse tree with this listener, then we see the following output:

```

entering root
entering sum
entering primitive
terminal 54
exiting primitive
terminal +
entering primitive
terminal (
entering sum
entering primitive
terminal 2
exiting primitive
terminal +
entering primitive
terminal 89
exiting primitive
exit sum
terminal )
exiting primitive
exit sum
terminal <EOF>
exiting root

```



Compare this printout with the parse tree shown at the right, and you'll see that the `ParseTreeWalker` is stepping through the nodes of the tree in order, from parents to children, and from left to right through the siblings.

Constructing an abstract syntax tree

We need to convert the parse tree into a recursive data type. Here's the definition of the recursive data type that we're going to use to represent integer arithmetic expressions:

```

IntegerExpression = Number(n:int)
                  + Plus(left:IntegerExpression, right:IntegerExpression)

```

If this syntax is mysterious, review recursive data type definitions ([../16-recursive-data-types/recursive/#recursive_datatype_definitions](#)) .

When a recursive data type represents a language this way, it is often called an *abstract syntax tree* . A `IntegerExpression` (<https://github.com/mit6005/sp16-ex18-parser-generators/blob/master/src/intexpr/IntegerExpression.java>) value captures the important features of the expression – its grouping and the integers in it – while omitting unnecessary details of the sequence of characters that created it.

By contrast, the parse tree that we just generated with the `IntegerExpression` parser is a *concrete syntax tree* . It's called concrete, rather than abstract, because it contains more details about how the expression is represented in actual characters. For example, the strings `2+2` , `((2)+(2))` , and `0002+0002` would each produce a different concrete syntax tree, but these trees would all correspond to the same abstract `IntegerExpression` value: `Plus(Number(2), Number(2))` .

Now we create a listener that constructs a `IntegerExpression` tree while it's walking over the parse tree. Each parse tree node will correspond to a `IntegerExpression` variant: `sum` nodes will create `Plus` (<https://github.com/mit6005/sp16-ex18-parser-generators/blob/master/src/intexpr/IntegerExpression.java>)

generators/blob/master/src/intexpr/IntegerExpression.java#L36) objects, and primitive nodes (that matched the NUMBER terminal) will create Number (<https://github.com/mit6005/sp16-ex18-parser-generators/blob/master/src/intexpr/IntegerExpression.java#L12>) objects.

Some primitive nodes are parenthesized subexpressions, not numbers. For these nodes, our listener will construct no new IntegerExpression object at all. Parentheses are concrete syntax whose meaning is captured in the abstract syntax tree by the structure of Plus and Number objects, and we already have a Plus to represent the sum inside the parentheses.

Whenever the walker exits each node of the parse tree, we have walked over the entire subtree under that node, so we create the next IntegerExpression object at exit time. But we have to keep track of all the children that were created during the walk over that subtree. We use a stack to store them.

Here's the code:

```

/** Make a IntegerExpression value from a parse tree. */
class MakeIntegerExpression implements IntegerExpressionListener {
    private Stack<IntegerExpression> stack = new Stack<>();
    // Invariant: stack contains the IntegerExpression value of each parse
    // subtree that has been fully-walked so far, but whose parent has not yet
    // been exited by the walk. The stack is ordered by recency of visit, so that
    // the top of the stack is the IntegerExpression for the most recently walked
    // subtree.
    //
    // At the start of the walk, the stack is empty, because no subtrees have
    // been fully walked.
    //
    // Whenever a node is exited by the walk, the IntegerExpression values of its
    // children are on top of the stack, in order with the last child on top. To
    // preserve the invariant, we must pop those child IntegerExpression values
    // from the stack, combine them with the appropriate IntegerExpression
    // producer, and push back an IntegerExpression value representing the entire
    // subtree under the node.
    //
    // At the end of the walk, after all subtrees have been walked and the root
    // has been exited, only the entire tree satisfies the invariant's
    // "fully walked but parent not yet exited" property, so the top of the stack
    // is the IntegerExpression of the entire parse tree.

    /**
     * Returns the expression constructed by this listener object.
     * Requires that this listener has completely walked over an IntegerExpression
     */
    public IntegerExpression getExpression() {
        return stack.get(0);
    }

    @Override public void exitRoot(IntegerExpressionParser.RootContext context) {
        // do nothing, root has only one child so its value is
        // already on top of the stack
    }

    @Override public void exitSum(IntegerExpressionParser.SumContext context) {
        // matched the primitive ('+' primitive)* rule
        List<IntegerExpressionParser.PrimitiveContext> addends = context.primitive();
        assert stack.size() >= addends.size();

        // the pattern above always has at least 1 child;
        // pop the last child
        assert addends.size() > 0;
        IntegerExpression sum = stack.pop();

        // pop the older children, one by one, and add them on
        for (int i = 1; i < addends.size(); ++i) {
            sum = new Plus(stack.pop(), sum);
        }
    }
}

```

```

// the result is this subtree's IntegerExpression
stack.push(sum);
}

@Override public void exitPrimitive(IntegerExpressionParser.PrimitiveContext
context) {
    if (context.NUMBER() != null) {
        // matched the NUMBER alternative
        int n = Integer.valueOf(context.NUMBER().getText());
        IntegerExpression number = new Number(n);
        stack.push(number);
    } else {
        // matched the '(' sum ')' alternative
        // do nothing, because sum's value is already on the stack
    }
}

// don't need these here, so just make them empty implementations
@Override public void enterRoot(IntegerExpressionParser.RootContext context)
{ }
@Override public void enterSum(IntegerExpressionParser.SumContext context) {
}
@Override public void enterPrimitive(IntegerExpressionParser.PrimitiveContext
context) { }

@Override public void visitTerminal(TerminalNode terminal) { }
@Override public void enterEveryRule(ParserRuleContext context) { }
@Override public void exitEveryRule(ParserRuleContext context) { }
@Override public void visitErrorNode(ErrorNode node) { }
}

```

More about Antlr's parse-tree listeners can be found in Section 7.2 of the Definitive ANTLR 4 Reference (<https://www.google.com/search?q=The+Definitive+Antlr+4+Reference>) .

Handling errors

By default, Antlr parsers print errors to the console. In order to make the parser modular, however, we need to handle those errors differently. You can attach an `ErrorListener` to the lexer and parser in order to throw an exception when an error is encountered during parsing. The `Configuration.g4` file defines a method `reportErrorsAsExceptions()` which does this. So if you copy the technique used in this grammar file, you can call:

```

lexer.reportErrorsAsExceptions();
parser.reportErrorsAsExceptions();

```

right after you create the lexer and parser. Then when you call `parser.root()`, it will throw an exception as soon as it encounters something that it can't match.

This is a simplistic approach to handling errors. Antlr offers more sophisticated forms of error recovery as well. To learn more, see Chapter 9 in the Definitive Antlr 4 Reference (<https://www.google.com/search?q=The+Definitive+Antlr+4+Reference>) .

Summary

The topics of today's reading connect to our three properties of good software as follows:

- **Safe from bugs.** A grammar is a declarative specification for strings and streams, which can be implemented automatically by a parser generator. These specifications are often simpler, more direct, and less likely to be buggy than parsing code written by hand.
- **Easy to understand.** A grammar captures the shape of a sequence in a form that is compact and easier to understand than hand-written parsing code.
- **Ready for change.** A grammar can be easily edited, then run through a parser generator to regenerate the parsing code.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 19: Concurrency

Concurrency

Two Models for Concurrent Programming

Processes, Threads, Time-slicing

Shared Memory Example

Interleaving

Race Condition

Tweaking the Code Won't Help

Reordering

Message Passing Example

Concurrency is Hard to Test and Debug

Summary

Reading 19: Concurrency

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- The message passing and shared memory models of concurrency
- Concurrent processes and threads, and time slicing
- The danger of race conditions

Concurrency

Concurrency means multiple computations are happening at the same time. Concurrency is everywhere in modern programming, whether we like it or not:

- Multiple computers in a network
- Multiple applications running on one computer
- Multiple processors in a computer (today, often multiple processor cores on a single chip)

In fact, concurrency is essential in modern programming:

- Web sites must handle multiple simultaneous users.

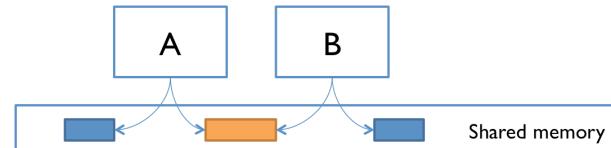
- Mobile apps need to do some of their processing on servers (“in the cloud”).
- Graphical user interfaces almost always require background work that does not interrupt the user. For example, Eclipse compiles your Java code while you’re still editing it.

Being able to program with concurrency will still be important in the future. Processor clock speeds are no longer increasing. Instead, we’re getting more cores with each new generation of chips. So in the future, in order to get a computation to run faster, we’ll have to split up a computation into concurrent pieces.

Two Models for Concurrent Programming

There are two common models for concurrent programming: *shared memory* and *message passing*.

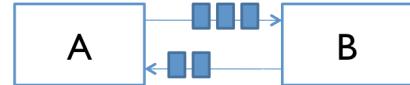
Shared memory. In the shared memory model of concurrency, concurrent modules interact by reading and writing shared objects in memory.



Examples of the shared-memory model:

- A and B might be two processors (or processor cores) in the same computer, sharing the same physical memory.
- A and B might be two programs running on the same computer, sharing a common filesystem with files they can read and write.
- A and B might be two threads in the same Java program (we’ll explain what a thread is below), sharing the same Java objects.

Message passing. In the message-passing model, concurrent modules interact by sending messages to each other through a communication channel. Modules send off messages, and incoming messages to each module are queued up for handling. Examples include:



- A and B might be two computers in a network, communicating by network connections.
- A and B might be a web browser and a web server – A opens a connection to B and asks for a web page, and B sends the web page data back to A.
- A and B might be an instant messaging client and server.
- A and B might be two programs running on the same computer whose input and output have been connected by a pipe, like `ls | grep` typed into a command prompt.

Processes, Threads, Time-slicing

The message-passing and shared-memory models are about how concurrent modules communicate. The concurrent modules themselves come in two different kinds: processes and threads.

Process. A process is an instance of a running program that is *isolated* from other processes on the same machine. In particular, it has its own private section of the machine’s memory.

The process abstraction is a *virtual computer*. It makes the program feel like it has the entire machine to itself – like a fresh computer has been created, with fresh memory, just to run that program.

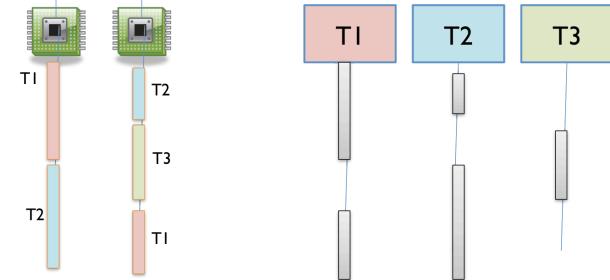
Just like computers connected across a network, processes normally share no memory between them. A process can’t access another process’s memory or objects at all. Sharing memory between processes is *possible* on most operating systems, but it needs special effort. By contrast, a new process is automatically ready for message passing, because it is created with standard input & output streams, which are the `System.out` and `System.in` streams you’ve used in Java.

Thread. A thread is a locus of control inside a running program. Think of it as a place in the program that is being run, plus the stack of method calls that led to that place (so the thread can go back up the stack when it reaches `return` statements).

Just as a process represents a virtual computer, the thread abstraction represents a *virtual processor*. Making a new thread simulates making a fresh processor inside the virtual computer represented by the process. This new virtual processor runs the same program and shares the same memory as other threads in the process.

Threads are automatically ready for shared memory, because threads share all the memory in the process. It takes special effort to get “thread-local” memory that’s private to a single thread. It’s also necessary to set up message-passing explicitly, by creating and using queue data structures. We’ll talk about how to do that in a future reading.

How can I have many concurrent threads with only one or two processors in my computer? When there are more threads than processors, concurrency is simulated by **time slicing**, which means that the processor switches between threads. The figure on the right shows how three threads T1, T2, and T3 might be time-sliced on a machine that has only two actual processors. In the figure, time proceeds downward, so at first one processor is running thread T1 and the other is running thread T2, and then the second processor switches to run thread T3. Thread T2 simply pauses, until its next time slice on the same processor or another processor.



On most systems, time slicing happens unpredictably and nondeterministically, meaning that a thread may be paused or resumed at any time.

In the Java Tutorials, read:

- **Processes & Threads**
[\(just 1 page\)](https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html)
- **Defining and Starting a Thread**
[\(just 1 page\)](https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html)

The second Java Tutorials reading shows two ways to create a thread.

- Never use their second way (subclassed `Thread`).
- Always implement the `Runnable` [interface](https://docs.oracle.com/javase/8/docs/api/?java/lang/Runnable.html) and use the `new Thread(...)` constructor.

Their example declares a named class that implements `Runnable`:

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
}
// ... in the main method:
new Thread(new HelloRunnable()).start();
```

A very common idiom is starting a thread with an anonymous [Runnable](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html), which eliminates the named class:

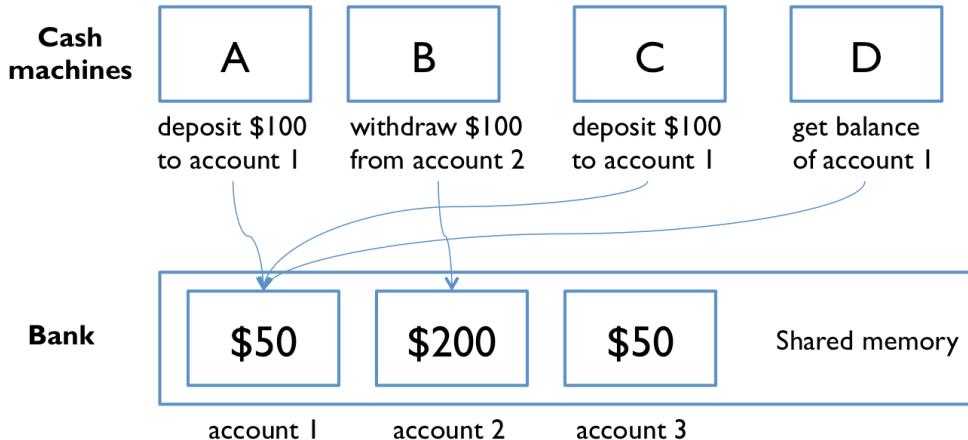
```
new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello from a thread!");
    }
}).start();
```

Read: **using an anonymous Runnable to start a thread (anonymous-runnable/)**

Shared Memory Example

Let's look at an example of a shared memory system. The point of this example is to show that concurrent programming is hard, because it can have subtle bugs.

Imagine that a bank has cash machines that use a shared memory model, so all the cash machines can read and write the same account objects in memory.



To illustrate what can go wrong, let's simplify the bank down to a single account, with a dollar balance stored in the `balance` variable, and two operations `deposit` and `withdraw` that simply add or remove a dollar:

```
// suppose all the cash machines share a single bank account
private static int balance = 0;

private static void deposit() {
    balance = balance + 1;
}
private static void withdraw() {
    balance = balance - 1;
}
```

Customers use the cash machines to do transactions like this:

```
deposit(); // put a dollar in
withdraw(); // take it back out
```

In this simple example, every transaction is just a one dollar deposit followed by a one-dollar withdrawal, so it should leave the balance in the account unchanged. Throughout the day, each cash machine in our network is processing a sequence of deposit/withdraw transactions.

```
// each ATM does a bunch of transactions that
// modify balance, but leave it unchanged afterward
private static void cashMachine() {
    for (int i = 0; i < TRANSACTIONS_PER_MACHINE; ++i) {
        deposit(); // put a dollar in
        withdraw(); // take it back out
    }
}
```

So at the end of the day, regardless of how many cash machines were running, or how many transactions we processed, we should expect the account balance to still be 0.

But if we run this code, we discover frequently that the balance at the end of the day is *not* 0. If more than one `cashMachine()` call is running at the same time – say, on separate processors in the same computer – then `balance` may not be zero at the end of the day. Why not?

Interleaving

Here's one thing that can happen. Suppose two cash machines, A and B, are both working on a deposit at the same time. Here's how the `deposit()` step typically breaks down into low-level processor instructions:

get balance (balance=0)

add 1

write back the result (balance=1)

When A and B are running concurrently, these low-level instructions interleave with each other (some might even be simultaneous in some sense, but let's just worry about interleaving for now):

A	B
A get balance (balance=0)	
A add 1	
A write back the result (balance=1)	
	B get balance (balance=1)
	B add 1
	B write back the result (balance=2)

This interleaving is fine – we end up with balance 2, so both A and B successfully put in a dollar. But what if the interleaving looked like this:

A	B
A get balance (balance=0)	
	B get balance (balance=0)
A add 1	
	B add 1

A**B**

A write back the result (balance=1)

B write back the result (balance=1)

The balance is now 1 – A's dollar was lost! A and B both read the balance at the same time, computed separate final balances, and then raced to store back the new balance – which failed to take the other's deposit into account.

Race Condition

This is an example of a **race condition**. A race condition means that the correctness of the program (the satisfaction of postconditions and invariants) depends on the relative timing of events in concurrent computations A and B. When this happens, we say "A is in a race with B."

Some interleavings of events may be OK, in the sense that they are consistent with what a single, nonconcurrent process would produce, but other interleavings produce wrong answers – violating postconditions or invariants.

Tweaking the Code Won't Help

All these versions of the bank-account code exhibit the same race condition:

```
// version 1
private static void deposit() { balance = balance + 1; }
private static void withdraw() { balance = balance - 1; }

// version 2
private static void deposit() { balance += 1; }
private static void withdraw() { balance -= 1; }

// version 3
private static void deposit() { ++balance; }
private static void withdraw() { --balance; }
```

You can't tell just from looking at Java code how the processor is going to execute it. You can't tell what the indivisible operations – the atomic operations – will be. It isn't atomic just because it's one line of Java. It doesn't touch balance only once just because the balance identifier occurs only once in the line. The Java compiler, and in fact the processor itself, makes no commitments about what low-level operations it will generate from your code. In fact, a typical modern Java compiler produces exactly the same code for all three of these versions!

The key lesson is that you can't tell by looking at an expression whether it will be safe from race conditions.

Read: **Thread Interference**

(<https://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html>) (just 1 page)

Reordering

It's even worse than that, in fact. The race condition on the bank account balance can be explained in terms of different interleavings of sequential operations on different processors. But in fact, when you're using multiple variables and multiple processors, you can't even count on changes to those variables

appearing in the same order.

Here's an example. Note that it uses a loop that continuously checks for a concurrent condition; this is called busy waiting (https://en.wikipedia.org/wiki/Busy_waiting) and it is not a good pattern. In this case, the code is also broken:

```
private boolean ready = false;
private int answer = 0;

// computeAnswer runs in one thread
private void computeAnswer() {
    answer = 42;
    ready = true;
}

// useAnswer runs in a different thread
private void useAnswer() {
    while (!ready) {
        Thread.yield();
    }
    if (answer == 0) throw new RuntimeException("answer wasn't ready!");
}
```

We have two methods that are being run in different threads. `computeAnswer` does a long calculation, finally coming up with the answer 42, which it puts in the `answer` variable. Then it sets the `ready` variable to true, in order to signal to the method running in the other thread, `useAnswer`, that the answer is ready for it to use. Looking at the code, `answer` is set before `ready` is set, so once `useAnswer` sees `ready` as true, then it seems reasonable that it can assume that the `answer` will be 42, right? Not so.

The problem is that modern compilers and processors do a lot of things to make the code fast. One of those things is making temporary copies of variables like `answer` and `ready` in faster storage (registers or caches on a processor), and working with them temporarily before eventually storing them back to their official location in memory. The storeback may occur in a different order than the variables were manipulated in your code. Here's what might be going on under the covers (but expressed in Java syntax to make it clear). The processor is effectively creating two temporary variables, `tmp_r` and `tmp_a`, to manipulate the fields `ready` and `answer`:

```
private void computeAnswer() {
    boolean tmp_r = ready;
    int tmp_a = answer;

    tmp_a = 42;
    tmp_r = true;

    ready = tmp_r;
        // <-- what happens if useAnswer() interleaves here?
        // ready is set, but answer isn't.
    answer = tmp_a;
}
```

Message Passing Example

Now let's look at the message-passing approach to our bank account example.

Now not only are the cash machine modules, but the accounts are modules, too. Modules interact by sending messages to each other.

Incoming requests are placed in a queue to be handled one at a time. The sender doesn't stop working while waiting for an

answer to its request. It handles more requests from its own queue. The reply to its request eventually comes back as another message.

Unfortunately, message passing doesn't eliminate the possibility of race conditions. Suppose each account supports `get-balance` and `withdraw` operations, with corresponding messages. Two users, at cash machines A and B, are both trying to withdraw a dollar from the same account. They check the balance first to make sure they never withdraw more than the account holds, because overdrafts trigger big bank penalties:

```
get-balance
if balance >= 1 then withdraw 1
```

The problem is again interleaving, but this time interleaving of the *messages* sent to the bank account, rather than the *instructions* executed by A and B. If the account starts with a dollar in it, then what interleaving of messages will fool A and B into thinking they can both withdraw a dollar, thereby overdrawning the account?

One lesson here is that you need to carefully choose the operations of a message-passing model. `withdraw-if-sufficient-funds` would be a better operation than just `withdraw`.

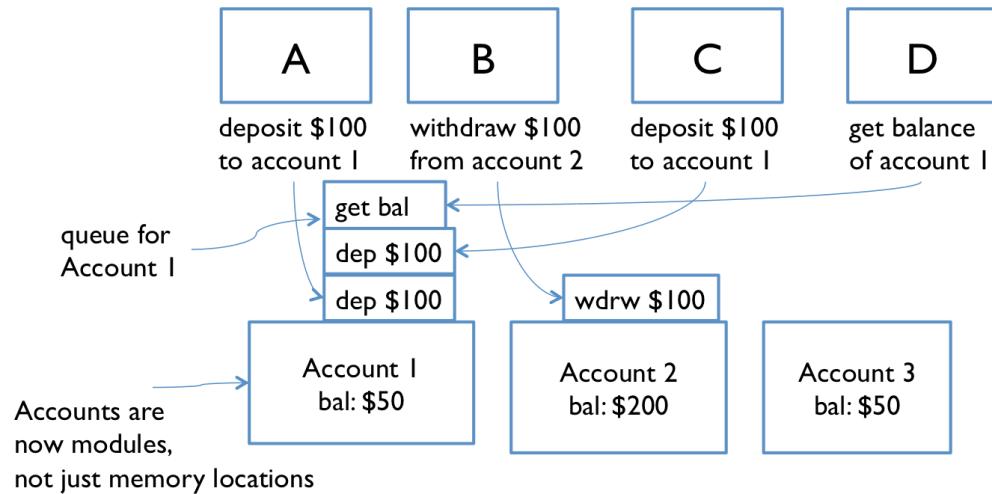
Concurrency is Hard to Test and Debug

If we haven't persuaded you that concurrency is tricky, here's the worst of it. It's very hard to discover race conditions using testing. And even once a test has found a bug, it may be very hard to localize it to the part of the program causing it.

Concurrency bugs exhibit very poor reproducibility. It's hard to make them happen the same way twice. Interleaving of instructions or messages depends on the relative timing of events that are strongly influenced by the environment. Delays can be caused by other running programs, other network traffic, operating system scheduling decisions, variations in processor clock speed, etc. Each time you run a program containing a race condition, you may get different behavior.

These kinds of bugs are *heisenbugs*, which are nondeterministic and hard to reproduce, as opposed to a *bohrbug*, which shows up repeatedly whenever you look at it. Almost all bugs in sequential programming are bohrbugs.

A heisenbug may even disappear when you try to look at it with `println` or debugger! The reason is that printing and debugging are so much slower than other operations, often 100-1000x slower, that they dramatically change the timing of operations, and the interleaving. So inserting a simple print statement into the `cashMachine()`:



```
private static void cashMachine() {  
    for (int i = 0; i < TRANSACTIONS_PER_MACHINE; ++i) {  
        deposit(); // put a dollar in  
        withdraw(); // take it back out  
        System.out.println(balance); // makes the bug disappear!  
    }  
}
```

...and suddenly the balance is always 0, as desired, and the bug appears to disappear. But it's only masked, not truly fixed. A change in timing somewhere else in the program may suddenly make the bug come back.

Concurrency is hard to get right. Part of the point of this reading is to scare you a bit. Over the next several readings, we'll see principled ways to design concurrent programs so that they are safer from these kinds of bugs.

Summary

- Concurrency: multiple computations running simultaneously
- Shared-memory & message-passing paradigms
- Processes & threads
 - Process is like a virtual computer; thread is like a virtual processor
- Race conditions
 - When correctness of result (postconditions and invariants) depends on the relative timing of events

These ideas connect to our three key properties of good software mostly in bad ways. Concurrency is necessary but it causes serious problems for correctness. We'll work on fixing those problems in the next few readings.

- **Safe from bugs.** Concurrency bugs are some of the hardest bugs to find and fix, and require careful design to avoid.
- **Easy to understand.** Predicting how concurrent code might interleave with other concurrent code is very hard for programmers to do. It's best to design your code in such a way that programmers don't have to think about interleaving at all.
- **Ready for change.** Not particularly relevant here.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 1: Static Checking

Hailstone Sequence

Computing Hailstones

Types

Static Typing

Static Checking, Dynamic Checking, No Checking

Surprise: Primitive Types Are Not True Numbers

Arrays and Collections

Iterating

Methods

Mutating Values vs. Reassigning Variables

Documenting Assumptions

Hacking vs. Engineering

The Goal of 6.005

Why we use Java in this course

Summary

Reading 1: Static Checking

Objectives for Today's Class

Today's class has two topics:

- static typing
- the big three properties of good software

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
 $2^n, 2^{n-1}, \dots, 4, 2, 1$ 
5, 16, 8, 4, 2, 1
7, 22, 11, 34, 17, 52, 6, 13, 40, ...? (where does this stop?)

```

Because of the odd-number rule, the sequence may bounce up and down before decreasing to 1. It's conjectured that all hailstones eventually fall to the ground – i.e., the hailstone sequence reaches 1 for all starting n – but that's still an open question (https://en.wikipedia.org/wiki/Collatz_conjecture). 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 some code for computing and printing the hailstone sequence for some starting n . We'll write Java and Python side by side for comparison:

```

// Java
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);

```

```

# Python
n = 3
while n != 1:
    print n
    if n % 2 == 0:
        n = n / 2
    else:
        n = 3 * n + 1
print 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.
- Java requires semicolons at the ends of statements. The extra punctuation can be a pain, but it also gives you more freedom in how you organize your code – you can split a statement into multiple lines for more readability.
- Java requires parentheses around the conditions of the `if` and `while`.
- Java requires curly braces around blocks, instead of indentation. You should always 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.

Types

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 $\pm 2^{31}$, or roughly ± 2 billion)
- `long` (for larger integers up to $\pm 2^{63}$)

- `boolean` (for true or false)
- `double` (for floating-point numbers, which represent a subset of the real numbers)
- `char` (for single characters like '`'A'`' and '`'$'`)

Java also has **object types**, for example:

- `String` represents a sequence of characters, like a Python string.
- `BigInteger` represents an integer of arbitrary size, so it acts like a Python integer.

By Java convention, primitive types are lowercase, while object types start with a capital letter.

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 no matter how they're written. Here are three different syntaxes for an operation in Python or Java:

- As an *infix, prefix, or postfix operator*. For example, `a + b` invokes the operation `+ : int × int → int`.
- As a *method of an object*. For example, `bigint1.add(bigint2)` calls the operation `add : BigInteger × BigInteger → BigInteger`.
- As a *function*. For example, `Math.sin(theta)` calls the operation `sin : double → double`. Here, `Math` is not an object. It's the class that contains the `sin` function.

Contrast Java's `str.length()` with Python's `len(str)`. It's the same operation in both languages – a function that takes a string and returns its length – but it just uses 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. Methods can also be overloaded. Most programming languages have some degree of overloading.

Static Typing

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 `int`s, 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 while you're still typing.

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

Static typing is a particular kind of **static checking**, which means checking for bugs at compile time. 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 typing 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"
```

that tries to multiply two strings, then static typing will catch this error while you're still programming, rather than waiting until the line is reached during execution.

Static Checking, Dynamic Checking, No Checking

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

- **Static checking**: the bug is found automatically before the program even runs.
- **Dynamic checking**: the bug is found automatically when the code is executed.

- **No checking** : the language doesn't help you find the error at all. You have to watch for it yourself, or end up with wrong answers.

Needless to say, catching a bug statically is better than catching it dynamically, and catching it dynamically is better than not catching it at all.

Here are some rules of thumb for what errors you can expect to be caught at each of these times.

Static checking can catch:

- syntax errors, like extra punctuation or spurious words. Even dynamically-typed languages like Python do this kind of static checking. If you have an indentation error in your Python program, you'll find out before the program starts running.
- wrong names, like `Math.sine(2)` . (The right name is `sin` .)
- wrong number of arguments, like `Math.sin(30, 20)` .
- wrong argument types, like `Math.sin("30")` .
- wrong return types, like `return "30";` from a function that's declared to return an `int` .

Dynamic checking can catch:

- illegal argument values. For example, the integer expression `x/y` is only erroneous when `y` is actually zero; otherwise it works. So in this expression, divide-by-zero is not a static error, but a dynamic error.
- unrepresentable return values, i.e., when the specific return value can't be represented in the type.
- out-of-range indexes, e.g., using a negative or too-large index on a string.
- calling a method on a `null` object reference (`null` is like Python `None`).

Static checking tends to be about types, errors that are independent of the specific value that a variable has. A type is a set of values. Static typing guarantees that a variable will have *some* value from that set, but we don't know until runtime exactly which value it has. So if the error would be caused only by certain values, like divide-by-zero or index-out-of-range then the compiler won't raise a static error about it.

Dynamic checking, by contrast, tends to be about errors caused by specific values.

Surprise: Primitive Types Are Not True Numbers

One trap in Java – and many other programming languages – is that its primitive numeric types have corner cases that do not behave like the integers and real numbers we're used to. As a result, some errors that really should be dynamically checked are not checked at all. Here are the traps:

- **Integer division** . `5/2` does not return a fraction, it returns a truncated integer. So this is an example of where what we might have hoped would be a dynamic error (because a fraction isn't representable as an integer) frequently produces the wrong answer instead.
- **Integer overflow** . The `int` and `long` types are actually finite sets of integers, with maximum and minimum values. What happens when you do a computation whose answer is too positive or too negative to fit in that finite range? The computation quietly *overflows* (wraps around), and returns an integer from somewhere in the legal range but not the right answer.
- **Special values in float and doubles** . The `float` and `double` types have several special values that aren't real numbers: `NaN` (which stands for "Not a Number"), `POSITIVE_INFINITY` , and `NEGATIVE_INFINITY` . So operations that you'd expect to produce dynamic errors, like dividing by zero or taking the square root of a negative number, produce one of these special values instead. If you keep computing with it, you'll end up with a bad final answer.

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`
- length: `a.length` (note that this is different syntax from `String.length()` – `a.length` is not a method call, so you don't put parentheses after it)

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]; // ===== DANGER WILL ROBINSON
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` that turned out to have a very long hailstone sequence? It wouldn't fit in a length-100 array. 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. Here's how we can declare a `List` variable and make a list value:

```
List<Integer> list = new ArrayList<Integer>();
```

And here are some of its operations:

- indexing: `list.get(2)`
- assignment: `list.set(2, 0)`
- length: `list.size()`

Note that `List` is an interface, a type that can't be constructed directly with `new`, but that instead specifies the operations that a `List` must provide. We'll talk about this notion in a future class 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, which you can find by searching the web for "Java 8 API". Get to know the Java API docs, they're your friend. ("API" means "application programmer interface," and is commonly used as a synonym for "library.")

Note also that we wrote `List<Integer>` instead of `List<int>`. Unfortunately we 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 (which are written in lowercase and often abbreviated, like `int`) has an equivalent object type (which is capitalized, and fully spelled out, like `Integer`). Java requires us to use these object type equivalents when we parameterize a type with angle brackets. But in other contexts, 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 Lists:

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

Not only simpler but safer too, because the `List` automatically enlarges itself to fit as many numbers as you add to it (until you run out of memory, of course).

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:

```
// find the maximum point of a hailstone sequence stored in list
int max = 0;
for (int x : list) {
    max = Math.max(x, max);
}
```

You can iterate through arrays as well as lists. The same code would work if the list were replaced by an array.

`Math.max()` is a handy function from the Java API. The `Math` class is full of useful functions like this – search for "java 8 Math" on the web to find its documentation.

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

```

Let's explain a few of the new things here.

`public` means that any code, anywhere in your program, can refer to the class or method. 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 class.

`static` means that the method doesn't take a self parameter – which in Java is implicit anyway, you won't ever see it as a method parameter. Static methods can't be called on an object. Contrast that with the `List add()` method or the `String length()` method, for example, which require an object to come first. Instead, the right way to call a static method uses the class name instead of an object reference:

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 positive, which is not captured by the type declaration but is very important for the caller to know.

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

Mutating Values vs. Reassigning Variables

The next reading will introduce *snapshot diagrams* to 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 in a future class 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> list = 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 (unfortunately) doesn't check automatically: that `n` must be positive.

Why do we need to write down our assumptions? Because 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. First persuading the compiler that your program is sensible – syntactically correct and type-correct. Then getting the logic right so that it gives the right results at runtime.
- communicating with other people. Making the program easy to understand, so that when somebody has to fix it, improve it, or adapt it in the future, they can do so.

Hacking vs. Engineering

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

- Bad: writing lots of code before testing any of it
- Bad: keeping all the details in your head, assuming you'll remember them forever, instead of writing them down in your code

- Bad: assuming that bugs will be nonexistent or else easy to find and fix

But software engineering is not hacking. Engineers are pessimists:

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

The Goal of 6.005

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 write 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.

There are other important properties of software (like performance, usability, security), and they may trade off against these three. But these are the Big Three that we care about in 6.005, and that software developers generally put foremost in the practice of building software. 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.

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, but other kinds of static checks too, like that your code returns values from methods declared to do so). 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 how in a safe, statically-checked language.

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 certainly 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, test frameworks, profilers, code coverage, style checkers). 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, having accumulated many features over the years. It's internally inconsistent (e.g. the `final` keyword means different things in different contexts, and the `static` keyword in Java has nothing to do with static checking). It's weighted with the baggage of older languages like C/C++ (the primitive types and the `switch` statement are good examples). It has no interpreter like Python's, where you can learn by playing with small bits of code.

But on the whole, Java is a reasonable choice of language right now to learn how to write code that is safe from bugs, easy to understand, and ready for change. And that's our goal.

Summary

The main idea we introduced today is **static checking**. Here's how this idea relates to the goals of the course:

- **Safe from bugs.** Static checking helps with safety by catching type errors and other bugs before runtime.
- **Easy to understand.** It helps with understanding, because types are explicitly stated in the code.
- **Ready for change.** Static checking makes it easier to change your code by identifying other places that need to change in tandem. For example, when you change the name or type of a variable, the compiler immediately displays errors at all the places where that variable is used, reminding you to update them as well.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 20: Thread Safety

What Threadsafe Means

Strategy 1: Confinement

Strategy 2: Immutability

Strategy 3: Using Threadsafe Data Types

How to Make a Safety Argument

Summary

Reading 20: Thread Safety

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

Recall race conditions: multiple threads sharing the same mutable variable without coordinating what they're doing. This is unsafe, because the correctness of the program may depend on accidents of timing of their low-level operations.

There are basically four ways to make variable access safe in shared-memory concurrency:

- **Confinement.** Don't share the variable between threads. This idea is called confinement, and we'll explore it today.
- **Immutability.** Make the shared data immutable. We've talked a lot about immutability already, but there are some additional constraints for concurrent programming that we'll talk about in this reading.
- **Threadsafe data type.** Encapsulate the shared data in an existing threadsafe data type that does the coordination for you. We'll talk about that today.
- **Synchronization.** Use synchronization to keep the threads from accessing the variable at the same time. Synchronization is what you need to build your own threadsafe data type.

We'll talk about the first three ways in this reading, along with how to make an argument that your code is threadsafe using those three ideas. We'll talk about the fourth approach, synchronization, in a later reading.

The material in this reading is inspired by an excellent book: Brian Goetz et al., *Java Concurrency in Practice* ([//jcip.net/](http://jcip.net/)) , Addison-Wesley, 2006.

What Threadsafe Means

A data type or static method is *threadsafe* if it behaves correctly when used from multiple threads, regardless of how those threads are executed, and without demanding additional coordination from the calling code.

- “behaves correctly” means satisfying its specification and preserving its rep invariant;
- “regardless of how threads are executed” means threads might be on multiple processors or timesliced on the same processor;
- “without additional coordination” means that the data type can’t put preconditions on its caller related to timing, like “you can’t call `get()` while `set()` is in progress.”

Remember `Iterator` ([//docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html](https://docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html)) ? It’s not threadsafe. `Iterator`’s specification says that you can’t modify a collection at the same time as you’re iterating over it. That’s a timing-related precondition put on the caller, and `Iterator` makes no guarantee to behave correctly if you violate it.

Strategy 1: Confinement

Our first way of achieving thread safety is *confinement*. Thread confinement is a simple idea: you avoid races on mutable data by keeping that data confined to a single thread. Don’t give any other threads the ability to read or write the data directly.

Since shared mutable data is the root cause of a race condition, confinement solves it by *not sharing* the mutable data.

Local variables are always thread confined. A local variable is stored in the stack, and each thread has its own stack. There may be multiple invocations of a method running at a time (in different threads or even at different levels of a single thread’s stack, if the method is recursive), but each of those invocations has its own private copy of the variable, so the variable itself is confined.

But be careful – the variable is thread confined, but if it’s an object reference, you also need to check the object it points to. If the object is mutable, then we want to check that the object is confined as well – there can’t be references to it that are reachable from any other thread.

Confinement is what makes the accesses to `n`, `i`, and `result` safe in code like this:

```

public class Factorial {

    /**
     * Computes n! and prints it on standard output.
     * @param n must be >= 0
     */
    private static void computeFact(final int n) {
        BigInteger result = new BigInteger("1");
        for (int i = 1; i <= n; ++i) {
            System.out.println("working on fact " + n);
            result = result.multiply(new BigInteger(String.valueOf(i)));
        }
        System.out.println("fact(" + n + ") = " + result);
    }

    public static void main(String[] args) {
        new Thread(new Runnable() { // create a thread using an
            public void run() { // anonymous Runnable
                computeFact(99);
            }
        }).start();
        computeFact(100);
    }
}

```

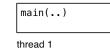
This code starts the thread for `computeFact(99)` with an anonymous [Runnable](https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html), a common idiom discussed in the previous reading ([./19-concurrency/anonymous-runnable/](#)).

Let's look at snapshot diagrams for this code. Hover or tap on each step to update the diagram:

1. When we start the program, we start with one thread running `main`
2. `main` creates a second thread using the anonymous `Runnable` idiom, and starts that thread.
3. At this point, we have two concurrent threads of execution. Their interleaving is unknown! But one *possibility* for the next thing that happens is that thread 1 enters `computeFact`.
4. Then, the next thing that *might* happen is that thread 2 also enters `computeFact`.

At this point, we see how **confinement** helps with thread safety: each execution of `computeFact` has its own `n`, `i`, and `result` variables. None of the objects they point to are mutable; if they were mutable, we would need to check that the objects are not aliased from other threads.

5. The `computeFact` computations proceed independently, updating their respective variables.



Avoid Global Variables

Unlike local variables, static variables are not automatically thread confined.

If you have static variables in your program, then you have to make an argument that only one thread will ever use them, and you have to document that fact clearly. Better, you should eliminate the static variables entirely.

Here's an example:

```
// This class has a race condition in it.
public class PinballSimulator {

    private static PinballSimulator simulator = null;
    // invariant: there should never be more than one PinballSimulator
    //             object created

    private PinballSimulator() {
        System.out.println("created a PinballSimulator object");
    }

    // factory method that returns the sole PinballSimulator object,
    // creating it if it doesn't exist
    public static PinballSimulator getInstance() {
        if (simulator == null) {
            simulator = new PinballSimulator();
        }
        return simulator;
    }
}
```

This class has a race in the `getInstance()` method – two threads could call it at the same time and end up creating two copies of the `PinballSimulator` object, which we don't want.

To fix this race using the thread confinement approach, you would specify that only a certain thread (maybe the “pinball simulation thread”) is allowed to call `PinballSimulator.getInstance()`. The risk here is that Java won't help you guarantee this.

In general, static variables are very risky for concurrency. They might be hiding behind an innocuous function that seems to have no side-effects or mutations. Consider this example:

```
// is this method threadsafe?
/**
 * @param x integer to test for primeness; requires x > 1
 * @return true if x is prime with high probability
 */
public static boolean isPrime(int x) {
    if (cache.containsKey(x)) return cache.get(x);
    boolean answer = BigInteger.valueOf(x).isProbablePrime(100);
    cache.put(x, answer);
    return answer;
}

private static Map<Integer, Boolean> cache = new HashMap<>();
```

This function stores the answers from previous calls in case they're requested again. This technique is called memoization (<https://en.wikipedia.org/wiki/Memoization>) , and it's a sensible optimization for slow functions like exact primality testing. But now the `isPrime` method is not safe to call from multiple threads, and its clients may not even realize it. The reason is that the `HashMap` referenced by the static variable `cache` is shared by all calls to `isPrime()` , and `HashMap` is not threadsafe. If multiple

threads mutate the map at the same time, by calling `cache.put()`, then the map can become corrupted in the same way that the bank account became corrupted in the last reading (./19-concurrency/#shared_memory_example). If you're lucky, the corruption may cause an exception deep in the hash map, like a `NullPointerException` or `IndexOutOfBoundsException`. But it also may just quietly give wrong answers, as we saw in the bank account example (./19-concurrency/#shared_memory_example).

Strategy 2: Immutability

Our second way of achieving thread safety is by using immutable references and data types. Immutability tackles the shared-mutable-data cause of a race condition and solves it simply by making the shared data *not mutable*.

Final variables are immutable references, so a variable declared final is safe to access from multiple threads. You can only read the variable, not write it. Be careful, because this safety applies only to the variable itself, and we still have to argue that the object the variable points to is immutable.

Immutable objects are usually also threadsafe. We say "usually" here because our current definition of immutability is too loose for concurrent programming. We've said that a type is immutable if an object of the type always represents the same abstract value for its entire lifetime. But that actually allows the type the freedom to mutate its rep, as long as those mutations are invisible to clients. We saw an example of this notion, called benevolent or beneficent mutation, when we looked at an immutable list that cached its length in a mutable field (./16-recursive-data-types/recursive/#tuning_the_rep) the first time the length was requested by a client. Caching is a typical kind of beneficent mutation.

For concurrency, though, this kind of hidden mutation is not safe. An immutable data type that uses beneficent mutation will have to make itself threadsafe using locks (the same technique required of mutable data types), which we'll talk about in a future reading.

Stronger definition of immutability

So in order to be confident that an immutable data type is threadsafe without locks, we need a stronger definition of immutability:

- no mutator methods
- all fields are private and final
- no representation exposure (./13-abstraction-functions-rep-invariants/#invariants)
- no mutation whatsoever of mutable objects in the rep – not even beneficent mutation (./16-recursive-data-types/recursive/#tuning_the_rep)

If you follow these rules, then you can be confident that your immutable type will also be threadsafe.

In the Java Tutorials, read:

- **A Strategy for Defining Immutable Objects**
[\(1 page\)](https://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html)

Strategy 3: Using Threadsafe Data Types

Our third major strategy for achieving thread safety is to store shared mutable data in existing threadsafe data types.

When a data type in the Java library is threadsafe, its documentation will explicitly state that fact. For example, here's what `StringBuffer` ([//docs.oracle.com/javase/8/docs/api/?java/lang/StringBuffer.html](https://docs.oracle.com/javase/8/docs/api/?java/lang/StringBuffer.html)) says:

[`StringBuffer` is] A thread-safe, mutable sequence of characters. A string buffer is like a `String`, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

String buffers are safe for use by multiple threads. The methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

This is in contrast to `StringBuilder` ([//docs.oracle.com/javase/8/docs/api/?java/lang/StringBuilder.html](https://docs.oracle.com/javase/8/docs/api/?java/lang/StringBuilder.html)) :

[`StringBuilder` is] A mutable sequence of characters. This class provides an API compatible with `StringBuffer`, but with no guarantee of synchronization. This class is designed for use as a drop-in replacement for `StringBuffer` in places where the string buffer was being used by a single thread (as is generally the case). Where possible, it is recommended that this class be used in preference to `StringBuffer` as it will be faster under most implementations.

It's become common in the Java API to find two mutable data types that do the same thing, one threadsafe and the other not. The reason is what this quote indicates: threadsafe data types usually incur a performance penalty compared to an unsafe type.

It's deeply unfortunate that `StringBuffer` and `StringBuilder` are named so similarly, without any indication in the name that thread safety is the crucial difference between them. It's also unfortunate that they don't share a common interface, so you can't simply swap in one implementation for the other for the times when you need thread safety. The Java collection interfaces do much better in this respect, as we'll see next.

Threadsafte Collections

The collection interfaces in Java – `List` , `Set` , `Map` – have basic implementations that are not threadsafe. The implementations of these that you've been used to using, namely `ArrayList` , `HashMap` , and `HashSet` , cannot be used safely from more than one thread.

Fortunately, just like the Collections API provides wrapper methods that make collections immutable, it provides another set of wrapper methods to make collections threadsafe, while still mutable.

These wrappers effectively make each method of the collection atomic with respect to the other methods. An **atomic action** effectively happens all at once – it doesn't interleave its internal operations with those of other actions, and none of the effects of the action are visible to other threads until the entire action is complete, so it never looks partially done.

Now we see a way to fix the `isPrime()` method we had earlier in the reading :

```
private static Map<Integer, Boolean> cache =
    Collections.synchronizedMap(new HashMap<>());
```

A few points here.

Don't circumvent the wrapper. Make sure to throw away references to the underlying non-threadsafte collection, and access it only through the synchronized wrapper. That happens automatically in the line of code above, since the new `HashMap` is passed only to `synchronizedMap()` and never stored

anywhere else. (We saw this same warning with the unmodifiable wrappers: the underlying collection is still mutable, and code with a reference to it can circumvent immutability.)

Iterators are still not threadsafe. Even though method calls on the collection itself (`get()`, `put()`, `add()`, etc.) are now threadsafe, iterators created from the collection are still not threadsafe. So you can't use `iterator()`, or the for loop syntax:

```
for (String s: lst) { ... } // not threadsafe, even if lst is a synchronized list
                           wrapper
```

The solution to this iteration problem will be to acquire the collection's lock when you need to iterate over it, which we'll talk about in a future reading.

Finally, **atomic operations aren't enough to prevent races**: the way that you use the synchronized collection can still have a race condition. Consider this code, which checks whether a list has at least one element and then gets that element:

```
if ( ! lst.isEmpty() ) { String s = lst.get(0); ... }
```

Even if you make `lst` into a synchronized list, this code still may have a race condition, because another thread may remove the element between the `isEmpty()` call and the `get()` call.

Even the `isPrime()` method still has potential races:

```
if (cache.containsKey(x)) return cache.get(x);
boolean answer = BigInteger.valueOf(x).isProbablePrime(100);
cache.put(x, answer);
```

The synchronized map ensures that `containsKey()`, `get()`, and `put()` are now atomic, so using them from multiple threads won't damage the rep invariant of the map. But those three operations can now interleave in arbitrary ways with each other, which might break the invariant that `isPrime` needs from the cache: if the cache maps an integer `x` to a value `f`, then `x` is prime if and only if `f` is true. If the cache ever fails this invariant, then we might return the wrong result.

So we have to argue that the races between `containsKey()`, `get()`, and `put()` don't threaten this invariant.

1. The race between `containsKey()` and `get()` is not harmful because we never remove items from the cache – once it contains a result for `x`, it will continue to do so.
2. There's a race between `containsKey()` and `put()`. As a result, it may end up that two threads will both test the primeness of the same `x` at the same time, and both will race to call `put()` with the answer. But both of them should call `put()` with the same answer, so it doesn't matter which one wins the race – the result will be the same.

The need to make these kinds of careful arguments about safety – even when you're using threadsafe data types – is the main reason that concurrency is hard.

In the Java Tutorials, read:

- **Wrapper Collections**
[\(1 page\)](https://docs.oracle.com/javase/tutorial/collections/implementations/wrapper.html)
- **Concurrent Collections**
[\(1 page\)](https://docs.oracle.com/javase/tutorial/essential/concurrency/collections.html)

How to Make a Safety Argument

We've seen that concurrency is hard to test and debug. So if you want to convince yourself and others that your concurrent program is correct, the best approach is to make an explicit argument that it's free from races, and write it down.

A safety argument needs to catalog all the threads that exist in your module or program, and the data that they use, and argue which of the four techniques you are using to protect against races for each data object or variable: confinement, immutability, threadsafe data types, or synchronization. When you use the last two, you also need to argue that all accesses to the data are appropriately atomic – that is, that the invariants you depend on are not threatened by interleaving. We gave one of those arguments for `isPrime` above.

Thread Safety Arguments for Data Types

Let's see some examples of how to make thread safety arguments for a data type. Remember our four approaches to thread safety: confinement, immutability, threadsafe data types, and synchronization. Since we haven't talked about synchronization in this reading, we'll just focus on the first three approaches.

Confinement is not usually an option when we're making an argument just about a data type, because you have to know what threads exist in the system and what objects they've been given access to. If the data type creates its own set of threads, then you can talk about confinement with respect to those threads. Otherwise, the threads are coming in from the outside, carrying client calls, and the data type may have no guarantees about which threads have references to what. So confinement isn't a useful argument in that case. Usually we use confinement at a higher level, talking about the system as a whole and arguing why we don't need thread safety for some of our modules or data types, because they won't be shared across threads by design.

Immutability is often a useful argument:

```
/** MyString is an immutable data type representing a string of characters. */
public class MyString {
    private final char[] a;
    // Thread safety argument:
    //   This class is threadsafe because it's immutable:
    //     - a is final
    //     - a points to a mutable char array, but that array is encapsulated
    //       in this object, not shared with any other object or exposed to a
    //       client
```

Here's another rep for `MyString` that requires a little more care in the argument:

```
/** MyString is an immutable data type representing a string of characters. */
public class MyString {
    private final char[] a;
    private final int start;
    private final int len;
    // Rep invariant:
    //   0 <= start <= a.length
    //   0 <= len <= a.length-start
    // Abstraction function:
    //   represents the string of characters a[start],...,a[start+length-1]
    // Thread safety argument:
    //   This class is threadsafe because it's immutable:
    //   - a, start, and len are final
    //   - a points to a mutable char array, which may be shared with other
    //     MyString objects, but they never mutate it
    //   - the array is never exposed to a client
```

Note that since this `MyString` rep was designed for sharing the array between multiple `MyString` objects, we have to ensure that the sharing doesn't threaten its thread safety. As long as it doesn't threaten the `MyString`'s immutability, however, we can be confident that it won't threaten the thread safety.

We also have to avoid rep exposure. Rep exposure is bad for any data type, since it threatens the data type's rep invariant. It's also fatal to thread safety.

Bad Safety Arguments

Here are some *incorrect* arguments for thread safety:

```
/** MyStringBuffer is a threadsafe mutable string of characters. */
public class MyStringBuffer {
    private String text;
    // Rep invariant:
    //   none
    // Abstraction function:
    //   represents the sequence text[0],...,text[text.length()-1]
    // Thread safety argument:
    //   text is an immutable (and hence threadsafe) String,
    //   so this object is also threadsafe
```

Why doesn't this argument work? `String` is indeed immutable and threadsafe; but the rep pointing to that string, specifically the `text` variable, is not immutable. `text` is not a final variable, and in fact it *can't* be final in this data type, because we need the data type to support insertion and deletion operations. So reads and writes of the `text` variable itself are not threadsafe. This argument is false.

Here's another broken argument:

```

public class Graph {
    private final Set<Node> nodes =
        Collections.synchronizedSet(new HashSet<>());
    private final Map<Node, Set<Node>> edges =
        Collections.synchronizedMap(new HashMap<>());
    // Rep invariant:
    //   for all x, y such that y is a member of edges.get(x),
    //       x, y are both members of nodes
    // Abstraction function:
    //   represents a directed graph whose nodes are the set of nodes
    //       and whose edges are the set (x,y) such that
    //           y is a member of edges.get(x)
    // Thread safety argument:
    //   - nodes and edges are final, so those variables are immutable
    //   and threadsafe
    //   - nodes and edges point to threadsafe set and map data types

```

This is a graph data type, which stores its nodes in a set and its edges in a map. (Quick quiz: is `Graph` a mutable or immutable data type? What do the `final` keywords have to do with its mutability?) `Graph` relies on other threadsafe data types to help it implement its rep – specifically the threadsafe `Set` and `Map` wrappers that we talked about above. That prevents some race conditions, but not all, because the graph's rep invariant includes a relationship *between* the node set and the edge map. All nodes that appear in the edge map also have to appear in the node set. So there may be code like this:

```

public void addEdge(Node from, Node to) {
    if ( ! edges.containsKey(from) ) {
        edges.put(from, Collections.synchronizedSet(new HashSet<>()));
    }
    edges.get(from).add(to);
    nodes.add(from);
    nodes.add(to);
}

```

This code has a race condition in it. There is a crucial moment when the rep invariant is violated, right after the `edges` map is mutated, but just before the `nodes` set is mutated. Another operation on the graph might interleave at that moment, discover the rep invariant broken, and return wrong results. Even though the threadsafe set and map data types guarantee that their own `add()` and `put()` methods are atomic and noninterfering, they can't extend that guarantee to *interactions* between the two data structures. So the rep invariant of `Graph` is not safe from race conditions. Just using immutable and threadsafe-mutable data types is not sufficient when the rep invariant depends on relationships *between* objects in the rep.

We'll have to fix this with synchronization, and we'll see how in a future reading.

Serializability

Look again at the code for the exercise above. We might also be concerned that `clear` and `insert` could interleave such that a client sees `clear` violate its postcondition.

Suppose two threads are sharing `MyStringBuffer` `sb` representing "z". They run

A	B
call <code>sb.clear()</code>	
	call <code>sb.insert(0, "a")</code>

clear and insert concurrently as shown on the right.

Thread A's assertion will fail, but not because clear violated its postcondition.

Indeed, when all the code in clear has finished running, the postcondition is satisfied.

A	B
— in clear : text = ""	— in insert : text = "" + "a" + "z"
— clear returns	— insert returns
	assert sb.toString() .equals("")

The real problem is that thread A has not anticipated possible interleaving between clear() and the assert . With any threadsafe mutable type where atomic mutators are called concurrently, *some* mutation has to “win” by being the last one applied. The result that thread A observed is identical to the execution below, where the mutators don't interleave at all:

A	B
call sb.clear()	
— in clear : text = ""	
— clear returns	
	call sb.insert(0, "a")
	— in insert : text = "" + "a" + "z"
	— insert returns
assert sb.toString() .equals("")	

What we demand from a threadsafe data type is that when clients call its atomic operations concurrently, the results are consistent with *some* sequential ordering of the calls. In this case, clearing and inserting, that means either clear -followed-by- insert , or insert -followed-by- clear . This property is called **serializability** (<https://en.wikipedia.org/wiki/Serializability>) : for any set of operations executed concurrently, the result (the values and state observable by clients) must be a result given by *some* sequential ordering of those operations.

Summary

This reading talked about three major ways to achieve safety from race conditions on shared mutable data:

- Confinement: not sharing the data.
- Immutability: sharing, but keeping the data immutable.
- Threadsafe data types: storing the shared mutable data in a single threadsafe datatype.

These ideas connect to our three key properties of good software as follows:

- **Safe from bugs.** We're trying to eliminate a major class of concurrency bugs, race conditions, and eliminate them by design, not just by accident of timing.
- **Easy to understand.** Applying these general, simple design patterns is far more understandable than a complex argument about which thread interleavings are possible and which are not.

- **Ready for change.** We're writing down these justifications explicitly in a thread safety argument, so that maintenance programmers know what the code depends on for its thread safety.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 21: Sockets & Networking

Client/server design pattern

Network sockets

I/O

Blocking

Using network sockets

Wire protocols

Testing client/server code

Summary

Reading 21: Sockets & Networking

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

In this reading we examine *client/server communication* over the network using the *socket abstraction*.

Network communication is inherently concurrent, so building clients and servers will require us to reason about their concurrent behavior and to implement them with thread safety. We must also design the *wire protocol* that clients and servers use to communicate, just as we design the operations that clients of an ADT use to work with it.

Some of the operations with sockets are *blocking* : they block the progress of a thread until they can return a result. Blocking makes writing some code easier, but it also foreshadows a new class of concurrency bugs we'll soon contend with in depth: deadlocks.

Client/server design pattern

In this reading (and in the problem set) we explore the **client/server design pattern** for communication with message passing.

In this pattern there are two kinds of processes: clients and servers. A client initiates the communication by connecting to a server. The client sends requests to the server, and the server sends replies back. Finally, the client disconnects. A server might handle connections from many clients concurrently, and clients might also connect to multiple servers.

Many Internet applications work this way: web browsers are clients for web servers, an email program like Outlook is a client for a mail server, etc.

On the Internet, client and server processes are often running on different machines, connected only by the network, but it doesn't have to be that way — the server can be a process running on the same machine as the client.

Network sockets

IP addresses

A network interface is identified by an IP address ([//en.wikipedia.org/wiki/IP_address](https://en.wikipedia.org/wiki/IP_address)) . IPv4 addresses are 32-bit numbers written in four 8-bit parts. For example (as of this writing):

- 18.9.22.69 is the IP address of a MIT web server. Every address whose first octet is 18 ([//en.wikipedia.org/wiki/List_of_assigned_/_8_IPv4_address_blocks](https://en.wikipedia.org/wiki/List_of_assigned_/_8_IPv4_address_blocks)) is on the MIT network.
- 18.9.25.15 is the address of a MIT incoming email handler.
- 173.194.123.40 is the address of a Google web server.
- 127.0.0.1 is the loopback ([//en.wikipedia.org/wiki/Loopback](https://en.wikipedia.org/wiki/Loopback)) or localhost ([//en.wikipedia.org/wiki/Localhost](https://en.wikipedia.org/wiki/Localhost)) address: it always refers to the local machine. Technically, any address whose first octet is 127 is a loopback address, but 127.0.0.1 is standard.

You can ask Google for your current IP address (<https://www.google.com/search?q=my+ip>) . In general, as you carry around your laptop, every time you connect your machine to the network it can be assigned a new IP address.

Hostnames

Hostnames ([//en.wikipedia.org/wiki/Hostname](https://en.wikipedia.org/wiki/Hostname)) are names that can be translated into IP addresses. A single hostname can map to different IP addresses at different times; and multiple hostnames can map to the same IP address. For example:

- web.mit.edu is the name for MIT's web server. You can translate this name to an IP address yourself using dig , host , or nslookup on the command line, e.g.:

```
$ dig +short web.mit.edu
18.9.22.69
```

- dmz-mailsec-scanner-4.mit.edu is the name for one of MIT's spam filter machines responsible for handling incoming email.
- google.com is exactly what you think it is. Try using one of the commands above to find google.com 's IP address. What do you see?
- localhost is a name for 127.0.0.1 . When you want to talk to a server running on your own machine, talk to localhost .

Translation from hostnames to IP addresses is the job of the Domain Name System (DNS) ([//en.wikipedia.org/wiki/Domain_Name_System](https://en.wikipedia.org/wiki/Domain_Name_System)) . It's super cool, but not part of our discussion today.

Port numbers

A single machine might have multiple server applications that clients wish to connect to, so we need a way to direct traffic on the same network interface to different processes.

Network interfaces have multiple ports ([//en.wikipedia.org/wiki/Port_\(computer_networking\)](https://en.wikipedia.org/wiki/Port_(computer_networking))) identified by a 16-bit number from 0 (which is reserved, so we effectively start at 1) to 65535.

A server process binds to a particular port — it is now **listening** on that port. Clients have to know which port number the server is listening on. There are some well-known ports ([//en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers#Well-known_ports](https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers#Well-known_ports)) which are reserved for system-level processes and provide standard ports for certain services. For example:

- Port 22 is the standard SSH port. When you connect to `athena.dialup.mit.edu` using SSH, the software automatically uses port 22.
- Port 25 is the standard email server port.
- Port 80 is the standard web server port. When you connect to the URL `http://web.mit.edu` in your web browser, it connects to `18.9.22.69` on port 80.

When the port is not a standard port, it is specified as part of the address. For example, the URL `http://128.2.39.10:9000` refers to port 9000 on the machine at `128.2.39.10`.

When a client connects to a server, that outgoing connection also uses a port number on the client's network interface, usually chosen at random from the available *non*-well-known ports.

Network sockets

A **socket** ([//en.wikipedia.org/wiki/Network_socket](https://en.wikipedia.org/wiki/Network_socket)) represents one end of the connection between client and server.

- A **listening socket** is used by a server process to wait for connections from remote clients.
In Java, use `ServerSocket` ([//docs.oracle.com/javase/8/docs/api/?java/net/ServerSocket.html](https://docs.oracle.com/javase/8/docs/api/?java/net/ServerSocket.html)) to make a listening socket, and use its `accept` ([//docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html#accept--](https://docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html#accept--)) method to listen to it.
- A **connected socket** can send and receive messages to and from the process on the other end of the connection. It is identified by both the local IP address and port number plus the remote address and port, which allows a server to differentiate between concurrent connections from different IPs, or from the same IP on different remote ports.
In Java, clients use a `Socket` ([//docs.oracle.com/javase/8/docs/api/?java/net/Socket.html](https://docs.oracle.com/javase/8/docs/api/?java/net/Socket.html)) constructor to establish a socket connection to a server. Servers obtain a connected socket as a `Socket` object returned from `ServerSocket.accept`.

I/O

Buffers

The data that clients and servers exchange over the network is sent in chunks. These are rarely just byte-sized chunks, although they might be. The sending side (the client sending a request or the server sending a response) typically writes a large chunk (maybe a whole string like "HELLO, WORLD!" or maybe 20 megabytes of video data). The network chops that chunk up into packets, and each packet is routed separately over the network. At the other end, the receiver reassembles the packets together into a stream of bytes.

The result is a bursty kind of data transmission — the data may already be there when you want to read them, or you may have to wait for them to arrive and be reassembled.

When data arrive, they go into a **buffer**, an array in memory that holds the data until you read it.

* see **What if Dr. Seuss Did Technical Writing?** ([//web.mit.edu/adorai/www/seuss-technical-writing.html](http://web.mit.edu/adorai/www/seuss-technical-writing.html)) , although the issue described in the first stanza is no longer relevant with the obsolescence of floppy disk drives

Streams

The data going into or coming out of a socket is a **stream** ([//en.wikipedia.org/wiki/Stream_\(computing\)](https://en.wikipedia.org/wiki/Stream_(computing))) of bytes.

In Java, `InputStream` ([//docs.oracle.com/javase/8/docs/api/?java/io/InputStream.html](https://docs.oracle.com/javase/8/docs/api/?java/io/InputStream.html)) objects represent sources of data flowing into your program. For example:

- Reading from a file on disk with a `FileInputStream` ([//docs.oracle.com/javase/8/docs/api/?java/io/FileInputStream.html](https://docs.oracle.com/javase/8/docs/api/?java/io/FileInputStream.html))
- User input from `System.in` ([//docs.oracle.com/javase/8/docs/api/java/lang/System.html#in](https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#in))
- Input from a network socket

`OutputStream` ([//docs.oracle.com/javase/8/docs/api/?java/io/OutputStream.html](https://docs.oracle.com/javase/8/docs/api/?java/io/OutputStream.html)) objects represent data sinks, places we can write data to. For example:

- `FileOutputStream` ([//docs.oracle.com/javase/8/docs/api/?java/io/FileOutputStream.html](https://docs.oracle.com/javase/8/docs/api/?java/io/FileOutputStream.html)) for saving to files
- `System.out` ([//docs.oracle.com/javase/8/docs/api/java/lang/System.html#out](https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#out)) is a `PrintStream` ([//docs.oracle.com/javase/8/docs/api/?java/io/PrintStream.html](https://docs.oracle.com/javase/8/docs/api/?java/io/PrintStream.html)) , an `OutputStream` that prints readable representations of various types
- Output to a network socket

In the Java Tutorials, read:

- **I/O Streams** ([//docs.oracle.com/javase/tutorial/essential/io/streams.html](https://docs.oracle.com/javase/tutorial/essential/io/streams.html)) up to and including *I/O from the Command Line* (8 pages)

With sockets, remember that the *output* of one process is the *input* of another process. If Alice and Bob have a socket connection, Alice has an output stream that flows to Bob's input stream, and *vice versa* .

Blocking

Blocking means that a thread waits (without doing further work) until an event occurs. We can use this term to describe methods and method calls: if a method is a **blocking method** , then a call to that method can **block** , waiting until some event occurs before it returns to the caller.

Socket input/output streams exhibit blocking behavior:

- When an incoming socket's buffer is empty, calling `read` blocks until data are available.
- When the destination socket's buffer is full, calling `write` blocks until space is available.

Blocking is very convenient from a programmer's point of view, because the programmer can write code as if the `read` (or `write`) call will always work, no matter what the timing of data arrival. If data (or for `write` , space) is already available in the buffer, the call might return very quickly. But if the `read` or `write` can't succeed, the call **blocks** . The operating system takes care of the details of delaying that thread until `read` or `write` can succeed.

Blocking happens throughout concurrent programming, not just in I/O ([//en.wikipedia.org/wiki/Input/output](https://en.wikipedia.org/wiki/Input/output)) (communication into and out of a process, perhaps over a network, or to/from a file, or with the user on the command line or a GUI, ...). Concurrent modules don't work in

lockstep, like sequential programs do, so they typically have to wait for each other to catch up when coordinated action is required.

We'll see in the next reading that this waiting gives rise to the second major kind of bug (the first was race conditions) in concurrent programming: **deadlock**, where modules are waiting for each other to do something, so none of them can make any progress. But that's for next time.

Using network sockets

Make sure you've read about streams at the Java Tutorial link above, then read about network sockets:

In the Java Tutorials, read:

- **All About Sockets** ([//docs.oracle.com/javase/tutorial/networking/sockets/index.html](https://docs.oracle.com/javase/tutorial/networking/sockets/index.html)) (4 pages)

This reading describes everything you need to know about creating server- and client-side sockets and writing to and reading from their I/O streams.

On the second page

The example uses a syntax we haven't seen: the try-with-resources ([//docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html](https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html)) statement. This statement has the form:

```
try (
    // create new objects here that require cleanup after being used,
    // and assign them to variables
) {
    // code here runs with those variables
    // cleanup happens automatically after the code completes
} catch(...) {
    // you can include catch clauses if the code might throw exceptions
}
```

On the last page

Notice how both `ServerSocket.accept()` and `in.readLine()` are *blocking*. This means that the server will need a *new thread* to handle I/O with each new client. While the client-specific thread is working with that client (perhaps blocked in a read or a write), another thread (perhaps the main thread) is blocked waiting to accept a new connection.

Unfortunately, their multithreaded Knock Knock Server implementation creates that new thread by *subclassed Thread*. That's *not* the recommended strategy. Instead, create a new class that implements `Runnable`, or use an anonymous `Runnable` ([./19-concurrency/anonymous-runnable/](#)) that calls a method where that client connection will be handled until it's closed. Don't use `extends Thread`. And while subclassing was popular when the Java API was designed, we don't discuss or recommend it at all because it has many downsides.

Wire protocols

Now that we have our client and server connected up with sockets, what do they pass back and forth over those sockets?

A **protocol** is a set of messages that can be exchanged by two communicating parties. A **wire protocol** in particular is a set of messages represented as byte sequences, like `hello world` and `bye` (assuming we've agreed on a way to encode those characters into bytes).

Most Internet applications use simple ASCII-based wire protocols. You can use a program called Telnet to check them out. For example:

HTTP

Hypertext Transfer Protocol (HTTP) ([//en.wikipedia.org/wiki/Hypertext_Transfer_Protocol](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol)) is the language of the World Wide Web. We already know that port 80 is the well-known port for speaking HTTP to web servers, so let's talk to one on the command line.

You'll be using Telnet on the problem set, so try these out now. User input is shown in **green**, and for input to the telnet connection, newlines (pressing enter) are shown with `↵`:

```
$ telnet www.eecs.mit.edu 80
Trying 18.62.0.96...
Connected to eecsweb.mit.edu.
Escape character is '^]'.
GET /↵
<!DOCTYPE html>
... lots of output ...
<title>Homepage | MIT EECS</title>
... lots more output ...
```

The `GET` command gets a web page. The `/` is the path of the page you want on the site. So this command fetches the page at `http://www.eecs.mit.edu:80/`. Since 80 is the default port for HTTP, this is equivalent to visiting `http://www.eecs.mit.edu` ([//www.eecs.mit.edu](http://www.eecs.mit.edu)) in your web browser. The result is HTML code that your browser renders to display the EECS homepage.

Internet protocols are defined by RFC specifications ([//en.wikipedia.org/wiki/Request_for_Comments](https://en.wikipedia.org/wiki/Request_for_Comments)) (RFC stands for “request for comment”, and some RFCs are eventually adopted as standards). RFC 1945 ([//tools.ietf.org/html/rfc1945](https://tools.ietf.org/html/rfc1945)) defined HTTP version 1.0, and was superseded by HTTP 1.1 in RFC 2616 ([//tools.ietf.org/html/rfc2616](https://tools.ietf.org/html/rfc2616)). So for many web sites, you might need to speak HTTP 1.1 if you want to talk to them. For example:

```
$ telnet web.mit.edu 80
Trying 18.9.22.69...
Connected to web.mit.edu.
Escape character is '^]'.
GET /aboutmit/ HTTP/1.1↵
Host: web.mit.edu↵
↵
HTTP/1.1 200 OK
Date: Tue, 31 Mar 2015 15:14:22 GMT
... more headers ...

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/T
... more HTML ...
<title>MIT – About</title>
... lots more HTML ...
```

This time, your request must end with a blank line. HTTP version 1.1 requires the client to specify some extra information (called headers) with the request, and the blank line signals the end of the headers.

You will also more than likely find that telnet does not exit after making this request — this time, the server keeps the connection open so you can make another request right away. To quit Telnet manually, type the escape character (probably `Ctrl -]`) to bring up the `telnet>` prompt, and type `quit` :

```
... lots more HTML ...
</html>
Ctrl-]↵
telnet> quit↵
Connection closed.
```

SMTP

Simple Mail Transfer Protocol (SMTP) ([//en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol](https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol)) is the protocol for sending email (different protocols are used for client programs that retrieve email from your inbox). Because the email system was designed in a time before spam, modern email communication is fraught with traps and heuristics designed to prevent abuse. But we can still try to speak SMTP. Recall that the well-known SMTP port is 25, and `dmz-mailsec-scanner-4.mit.edu` was the name of a MIT email handler.

You'll need to fill in `your-IP-address-here` and `your-username-here`, and the ↵ indicate newlines for clarity. This will only work if you're on MITnet, and even then your mail might be rejected for looking suspicious:

```
$ telnet dmz-mailsec-scanner-4.mit.edu 25
Trying 18.9.25.15...
Connected to dmz-mailsec-scanner-4.mit.edu.
Escape character is '^]'.
220 dmz-mailsec-scanner-4.mit.edu ESMTP Symantec Messaging Gateway
HELO your-IP-address-here↵
250 2.0.0 dmz-mailsec-scanner-4.mit.edu says HELO to your-ip-address:port
MAIL FROM: <your-username-here@mit.edu>↵
250 2.0.0 MAIL FROM accepted
RCPT TO: <your-username-here@mit.edu>↵
250 2.0.0 RCPT TO accepted
DATA↵
354 3.0.0 continue. finished with "\r\n.\r\n"
From: <your-username-here@mit.edu>↵
To: <your-username-here@mit.edu>↵
Subject: testing↵
This is a hand-crafted artisanal email.↵
.↵
250 2.0.0 OK 99/00-11111-22222222
QUIT↵
221 2.3.0 dmz-mailsec-scanner-4.mit.edu closing connection
Connection closed by foreign host.
```

SMTP is quite chatty in comparison to HTTP, providing some human-readable instructions like `continue`. `finished with "\r\n.\r\n"` to tell us how to terminate our message content.

Designing a wire protocol

When designing a wire protocol, apply the same rules of thumb you use for designing the operations of an abstract data type:

- Keep the number of different messages **small**. It's better to have a few commands and responses that can be combined rather than many complex messages.
- Each message should have a well-defined purpose and **coherent** behavior.
- The set of messages must be **adequate** for clients to make the requests they need to make and for servers to deliver the results.

Just as we demand representation independence from our types, we should aim for **platform-independence** in our protocols. HTTP can be spoken by any web server and any web browser on any operating system. The protocol doesn't say anything about how web pages are stored on disk, how they are prepared or generated by the server, what algorithms the client will use to render them, etc.

We can also apply the three big ideas in this class:

- **Safe from bugs**

- The protocol should be easy for clients and servers to generate and parse. Simpler code for reading and writing the protocol (whether written with a parser generator like ANTLR, with regular expressions, etc.) will have fewer opportunities for bugs.
- Consider the ways a broken or malicious client or server could stuff garbage data into the protocol to break the process on the other end.

Email spam is one example: when we spoke SMTP above, the mail server asked us to say who was sending the email, and there's nothing in SMTP to prevent us from lying outright. We've had to build systems on top of SMTP to try to stop spammers who lie about `From:` addresses.

Security vulnerabilities are a more serious example. For example, protocols that allow a client to send requests with arbitrary amounts of data require careful handling on the server to avoid running out of buffer space, or worse ([//en.wikipedia.org/wiki/Buffer_overflow](https://en.wikipedia.org/wiki/Buffer_overflow)) .

- **Easy to understand** : for example, choosing a text-based protocol means that we can debug communication errors by reading the text of the client/server exchange. It even allows us to speak the protocol "by hand" as we saw above.
- **Ready for change** : for example, HTTP includes the ability to specify a version number, so clients and servers can agree with one another which version of the protocol they will use. If we need to make changes to the protocol in the future, older clients or servers can continue to work by announcing the version they will use.

Serialization ([//en.wikipedia.org/wiki/Serialization](https://en.wikipedia.org/wiki/Serialization)) is the process of transforming data structures in memory into a format that can be easily stored or transmitted (not the same as serializability from *Thread Safety* ([../20-thread-safety/#serializability](#))). Rather than invent a new format for serializing your data between clients and servers, use an existing one. For example, JSON (JavaScript Object Notation) ([//en.wikipedia.org/wiki/JSON](https://en.wikipedia.org/wiki/JSON)) is a simple, widely-used format for serializing basic values, arrays, and maps with string keys.

Specifying a wire protocol

In order to precisely define for clients & servers what messages are allowed by a protocol, use a grammar.

For example, here is a very small part of the HTTP 1.1 request grammar from RFC 2616 section 5 ([//www.w3.org/Protocols/rfc2616/rfc2616-sec5.html](https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html)) :

```

request ::= request-line
          ((general-header | request-header | entity-header) CRLF)*
          CRLF
          message-body?
request-line ::= method SPACE request-uri SPACE http-version CRLF
method ::= "OPTIONS" | "GET" | "HEAD" | "POST" | ...
...
  
```

Using the grammar, we can see that in this example request from earlier:

```

GET /aboutmit/ HTTP/1.1
Host: web.mit.edu
  
```

- GET is the method : we're asking the server to get a page for us.
- /aboutmit/ is the request-uri : the description of what we want to get.
- HTTP/1.1 is the http-version .
- Host: web.mit.edu is some kind of header — we would have to examine the rules for each of the ...-header options to discover which one.
- And we can see why we had to end the request with a blank line: since a single request can have multiple headers that end in CRLF (newline), we have another CRLF at the end to finish the request .
- We don't have any message-body — and since the server didn't wait to see if we would send one, presumably that only applies for other kinds of requests.

The grammar is not enough: it fills a similar role to method signatures when defining an ADT. We still need the specifications:

- **What are the preconditions of a message?** For example, if a particular field in a message is a string of digits, is any number valid? Or must it be the ID number of a record known to the server? Under what circumstances can a message be sent? Are certain messages only valid when sent in a certain sequence?
- **What are the postconditions?** What action will the server take based on a message? What server-side data will be mutated? What reply will the server send back to the client?

Testing client/server code

Remember that concurrency is hard to test and debug (..19-concurrency/#concurrency_is_hard_to_test_and_debug) . We can't reliably reproduce race conditions, and the network adds a source of latency that is entirely beyond our control. You need to design for concurrency and argue carefully for the correctness of your code.

Separate network code from data structures and algorithms

Most of the ADTs in your client/server program don't need to rely on networking. Make sure you specify, test, and implement them as separate components that are safe from bugs, easy to understand, and ready for change — in part because they don't involve any networking code.

If those ADTs will need to be used concurrently from multiple threads (for example, threads handling different client connections), our next reading will discuss your options. Otherwise, use the thread safety strategies of confinement, immutability, and existing threadsafe data types (..20-thread-safety/) .

Separate socket code from stream code

A function or module that needs to read from and write to a socket may only need access to the input/output streams, not to the socket itself. This design allows you to test the module by connecting it to streams that don't come from a socket.

Two useful Java classes for this are `ByteArrayInputStream` ([//docs.oracle.com/javase/8/docs/api/?java/io/ByteArrayInputStream.html](https://docs.oracle.com/javase/8/docs/api/?java/io/ByteArrayInputStream.html)) and `ByteArrayOutputStream` ([//docs.oracle.com/javase/8/docs/api/?java/io/ByteArrayOutputStream.html](https://docs.oracle.com/javase/8/docs/api/?java/io/ByteArrayOutputStream.html)) . Suppose we want to test this method:

```
void upperCaseLine(BufferedReader input, PrintWriter output) throws IOException
    requires: input and output are open
    effects: attempts to read a line from input
            and attempts to write that line, in upper case, to output
```

The method is normally used with a socket:

```
Socket sock = ...

// read a stream of characters from the socket input stream
BufferedReader in = new BufferedReader(new InputStreamReader(sock.getInputStream()));
// write characters to the socket output stream
PrintWriter out = new PrintWriter(sock.getOutputStream(), true);

upperCaseLine(in, out);
```

If the case conversion is a function we implement, it should already be specified, tested, and implemented separately. But now we can now also test the read/write behavior of `upperCaseLine` :

```
// fixed input stream of "dog" (line 1) and "cat" (line 2)
String inString = "dog\ncat\n";
ByteArrayInputStream inBytes = new ByteArrayInputStream(inString.getBytes());
ByteArrayOutputStream outBytes = new ByteArrayOutputStream();

// read a stream of characters from the fixed input string
BufferedReader in = new BufferedReader(new InputStreamReader(inBytes));
// write characters to temporary storage
PrintWriter out = new PrintWriter(outBytes, true);

upperCaseLine(in, out);

// check that it read the expected amount of input
assertEquals("expected input line 2 remaining", "cat", in.readLine());
// check that it wrote the expected output
assertEquals("expected upper case of input line 1", "DOG\n", outBytes.toString());
```

In this test, `inBytes` and `outBytes` are **test stubs** ([..03-testing/#unit_testing_and_stubs](#)) . To isolate and test just `upperCaseLine` , we replace the components it normally depends on (input/output streams from a socket) with components that satisfy the same spec but have canned behavior: an input stream with fixed input, and an output stream that stores the output in memory.

Testing strategies for more complex modules might use a **mock object** to simulate the behavior of a real client or server by producing entire canned sequences of interaction and asserting the correctness of each message received from the other component.

Summary

In the *client/server design pattern*, concurrency is inevitable: multiple clients and multiple servers are connected on the network, sending and receiving messages simultaneously, and expecting timely replies. A server that *blocks* waiting for one slow client when there are other clients waiting to connect to it or to receive replies will not make those clients happy. At the same time, a server that performs incorrect computations or returns bogus results because of concurrent modification to shared mutable data by different clients will not make anyone happy.

All the challenges of making our multi-threaded code **safe from bugs**, **easy to understand**, and **ready for change** apply when we design network clients and servers. These processes run concurrently with one another (if on different machines), and any server that wants to talk to multiple clients concurrently (or a client that wants to talk to multiple servers) must manage that multi-threaded communication.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 22: Queues and Message-Passing

Two models for concurrency

Message passing with threads

Implementing message passing with queues

Stopping

Thread safety arguments with message passing

Summary

Reading 22: Queues and Message-Passing

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

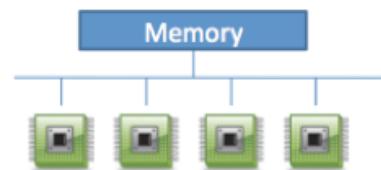
Objectives

After reading the notes and examining the code for this class, you should be able to use message passing (with synchronous queues) instead of shared memory for communication between threads.

Two models for concurrency

In our introduction to concurrency, we saw two models for concurrent programming (..19-concurrency/#two_models_for_concurrent_programming) : *shared memory* and *message passing* .

- In the **shared memory** model, concurrent modules interact by reading and writing shared mutable objects in memory. Creating multiple threads inside a single Java process is our primary example of shared-memory concurrency.
- In the **message passing** model, concurrent modules interact by sending immutable messages to one another over a communication channel. We've had one example of message passing so far: the client/server pattern (..21-sockets-networking/#clientserver_design_pattern) , in which clients and servers are concurrent processes, often on different machines, and the communication channel is a network socket (..21-sockets-networking/#network_sockets) .



The message passing model has several advantages over the shared memory model, which boil down to greater safety from bugs. In message-passing, concurrent modules interact *explicitly* , by passing messages through the communication channel, rather than *implicitly* through mutation of shared data.

The implicit interaction of shared memory can too easily lead to *inadvertent* interaction, sharing and manipulating data in parts of the program that don't know they're concurrent and aren't cooperating properly in the thread safety strategy. Message passing also shares only immutable objects (the messages) between modules, whereas shared memory *requires* sharing mutable objects, which we have already seen can be a source of bugs (./09-immutability/#risks_of_mutation).



We'll discuss in this reading how to implement message passing within a single process, as opposed to between processes over the network. We'll use **blocking queues** (an existing threadsafe type) to implement message passing between threads within a process.

Message passing with threads

We've previously talked about message passing between processes: clients and servers communicating over network sockets (./21-sockets-networking/#network_sockets). We can also use message passing between threads within the same process, and this design is often preferable to a shared memory design with locks, which we'll talk about in the next reading.

Use a synchronized queue for message passing between threads. The queue serves the same function as the buffered network communication channel in client/server message passing. Java provides the `BlockingQueue` ([//docs.oracle.com/javase/8/docs/api/?java/util/concurrent/BlockingQueue.html](https://docs.oracle.com/javase/8/docs/api/?java/util/concurrent/BlockingQueue.html)) interface for queues with blocking operations:

In an ordinary Queue ([//docs.oracle.com/javase/8/docs/api/?java/util/Queue.html](https://docs.oracle.com/javase/8/docs/api/?java/util/Queue.html)):

- `add(e)` adds element `e` to the end of the queue.
- `remove()` removes and returns the element at the head of the queue, or throws an exception if the queue is empty.

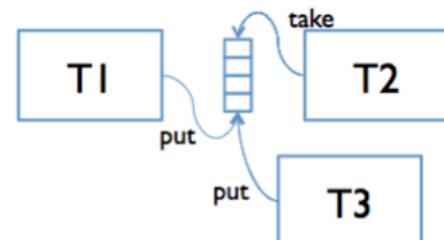
A `BlockingQueue` ([//docs.oracle.com/javase/8/docs/api/?java/util/concurrent/BlockingQueue.html](https://docs.oracle.com/javase/8/docs/api/?java/util/concurrent/BlockingQueue.html)) extends this interface:

additionaly supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

- `put(e)` blocks until it can add element `e` to the end of the queue (if the queue does not have a size bound, `put` will not block).
- `take()` blocks until it can remove and return the element at the head of the queue, waiting until the queue is non-empty.

When you are using a `BlockingQueue` for message passing between threads, make sure to use the `put()` and `take()` operations, not ~~`add()` and `remove()`~~.

Analogous to the client/server pattern for message passing over a network is the **producer-consumer design pattern** for message passing between threads. Producer threads and consumer threads share a synchronized queue. Producers put data or requests onto the queue, and consumers remove and process them. One or more producers and one or more consumers might all be adding and removing items from the same queue. This queue must be safe for concurrency.



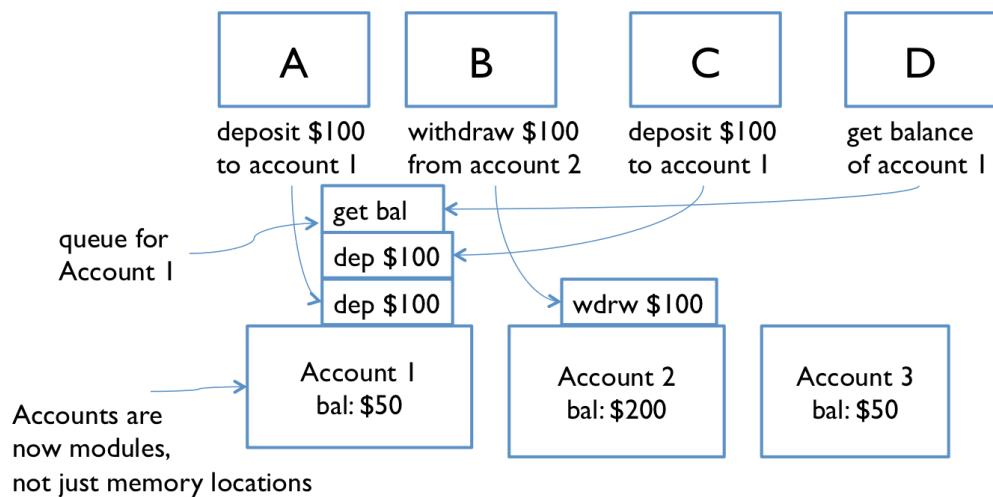
Java provides two implementations of `BlockingQueue`:

- `ArrayBlockingQueue` (<http://docs.oracle.com/javase/8/docs/api/?java/util/concurrent/ArrayBlockingQueue.html>) is a fixed-size queue that uses an array representation. Putting a new item on the queue will block if the queue is full.
- `LinkedBlockingQueue` (<http://docs.oracle.com/javase/8/docs/api/?java/util/concurrent/LinkedBlockingQueue.html>) is a growable queue using a linked-list representation. If no maximum capacity is specified, the queue will never fill up, so `put` will never block.

Unlike the streams of bytes sent and received by sockets, these synchronized queues (like normal collections classes in Java) can hold objects of an arbitrary type. Instead of designing a wire protocol, we must choose or design a type for messages in the queue. **It must be an immutable type.** And just as we did with operations on a threadsafe ADT or messages in a wire protocol, we must design our messages here to prevent race conditions and enable clients to perform the atomic operations they need.

Bank account example

Our first example of message passing was the bank account example (./19-



[concurrency/#message_passing_example](#).

Each cash machine and each account is its own module, and modules interact by sending messages to one another. Incoming messages arrive on a queue.

We designed messages for `get-balance` and `withdraw`, and said that each cash machine checks the account balance before withdrawing to prevent overdrafts:

```
get-balance
if balance >= 1 then withdraw 1
```

But it is still possible to interleave messages from two cash machines so they are both fooled into thinking they can safely withdraw the last dollar from an account with only \$1 in it.

We need to choose a better atomic operation: `withdraw-if-sufficient-funds` would be a better operation than just `withdraw`.

Implementing message passing with queues

You can see all the code for this example on GitHub: **square example** (<https://github.com/mit6005/sp16-ex22-square>). All the relevant parts are excerpted below.

Here's a message passing module for squaring integers:

`SquareQueue.java` line 6 (<https://github.com/mit6005/sp16-ex22-square/blob/master/src/square/SquareQueue.java#L6-L46>)

```
private final BlockingQueue<Integer> in;
private final BlockingQueue<SquareResult> out;
// Rep invariant: in, out != null

/** Make a new squarer.
 * @param requests queue to receive requests from
 * @param replies queue to send replies to */
public Squarer(BlockingQueue<Integer> requests,
               BlockingQueue<SquareResult> replies) {
    this.in = requests;
    this.out = replies;
}

/** Start handling squaring requests. */
public void start() {
    new Thread(new Runnable() {
        public void run() {
            while (true) {
                // TODO: we may want a way to stop the thread
                try {
                    // block until a request arrives
                    int x = in.take();
                    // compute the answer and send it back
                    int y = x * x;
                    out.put(new SquareResult(x, y));
                } catch (InterruptedException ie) {
                    ie.printStackTrace();
                }
            }
        }
    }).start();
}
```

Incoming messages to the `Squarer` are integers; the squarer knows that its job is to square those numbers, so no further details are required.

Outgoing messages are instances of `SquareResult` :

SquareResult.java line 48 (<https://github.com/mit6005/sp16-ex22-square/blob/master/src/square/SquareQueue.java#L48-L70>)

```

public class SquareResult {
    private final int input;
    private final int output;

    /** Make a new result message.
     * @param input input number
     * @param output square of input */
    public SquareResult(int input, int output) {
        this.input = input;
        this.output = output;
    }

    @Override public String toString() {
        return input + " ^2 = " + output;
    }
}

```

We would probably add additional observers to `SquareResult` so clients can retrieve the input number and output result.

Finally, here's a main method that uses the squarer:

SquareQueue.java line 77 (<https://github.com/mit6005/sp16-ex22-square/blob/master/src/square/SquareQueue.java#L77-L96>)

```

public static void main(String[] args) {
    BlockingQueue<Integer> requests = new LinkedBlockingQueue<>();
    BlockingQueue<SquareResult> replies = new LinkedBlockingQueue<>();

    Squarer squarer = new Squarer(requests, replies);
    squarer.start();

    try {
        // make a request
        requests.put(42);
        // ... maybe do something concurrently ...
        // read the reply
        System.out.println(replies.take());
    } catch (InterruptedException ie) {
        ie.printStackTrace();
    }
}

```

It should not surprise us that this code has a very similar flavor to the code for implementing message passing with sockets.

Stopping

What if we want to shut down the `Square` so it is no longer waiting for new inputs? In the client/server model, if we want the client or server to stop listening for our messages, we close the socket. And if we want the client or server to stop altogether, we can quit that process. But here, the square is just another thread in the *same* process, and we can't "close" a queue.

One strategy is a *poison pill*: a special message on the queue that signals the consumer of that message to end its work. To shut down the square, since its input messages are merely integers, we would have to choose a magic poison integer (everyone knows the square of 0 is 0 right? no one will need to ask for

the square of 0...) or use null (don't use null). Instead, we might change the type of elements on the requests queue to an ADT:

```
SquareRequest = IntegerRequest + StopRequest
```

with operations:

```
input : SquareRequest → int
shouldStop : SquareRequest → boolean
```

and when we want to stop the squarer, we enqueue a `StopRequest` where `shouldStop` returns `true`

For example, in `SquareRequest.start()` :

```
public void run() {
    while (true) {
        try {
            // block until a request arrives
            SquareRequest req = in.take();
            // see if we should stop
            if (req.shouldStop()) { break; }
            // compute the answer and send it back
            int x = req.input();
            int y = x * x;
            out.put(new SquareResult(x, y));
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
    }
}
```

It is also possible to *interrupt* a thread by calling its `interrupt()` method. If the thread is blocked waiting, the method it's blocked in will throw an `InterruptedException` (that's why we have to try-catch that exception almost any time we call a blocking method). If the thread was not blocked, an *interrupted* flag will be set. The thread must check for this flag to see whether it should stop working. For example:

```
public void run() {
    // handle requests until we are interrupted
    while ( ! Thread.interrupted() ) {
        try {
            // block until a request arrives
            int x = in.take();
            // compute the answer and send it back
            int y = x * x;
            out.put(new SquareResult(x, y));
        } catch (InterruptedException ie) {
            // stop
            break;
        }
    }
}
```

Thread safety arguments with message passing

A thread safety argument with message passing might rely on:

- **Existing threadsafe data types** for the synchronized queue. This queue is definitely shared and definitely mutable, so we must ensure it is safe for concurrency.
- **Immutability** of messages or data that might be accessible to multiple threads at the same time.
- **Confinement** of data to individual producer/consumer threads. Local variables used by one producer or consumer are not visible to other threads, which only communicate with one another using messages in the queue.
- **Confinement** of mutable messages or data that are sent over the queue but will only be accessible to one thread at a time. This argument must be carefully articulated and implemented. But if one module drops all references to some mutable data like a hot potato as soon as it puts them onto a queue to be delivered to another thread, only one thread will have access to those data at a time, precluding concurrent access.

In comparison to synchronization, message passing can make it easier for each module in a concurrent system to maintain its own thread safety invariants. We don't have to reason about multiple threads accessing shared data if the data are instead transferred between modules using a threadsafe communication channel.

Summary

- Rather than synchronize with locks, message passing systems synchronize on a shared communication channel, e.g. a stream or a queue.
- Threads communicating with blocking queues is a useful pattern for message passing within a single process.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 23: Locks and Synchronization

Introduction

Synchronization

Deadlock

Developing a threadsafe abstract data type

Locking

Monitor pattern

Thread safety argument with synchronization

Atomic operations

Designing a datatype for concurrency

Deadlock rears its ugly head

Goals of concurrent program design

Concurrency in practice

Summary

Reading 23: Locks and Synchronization

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- Understand how a lock is used to protect shared mutable data
- Be able to recognize deadlock and know strategies to prevent it
- Know the monitor pattern and be able to apply it to a data type

Introduction

Earlier, we defined thread safety (./20-thread-safety/#what_threadsafe_means) for a data type or a function as *behaving correctly when used from multiple threads, regardless of how those threads are executed, without additional coordination* .

Here's the general principle: **the correctness of a concurrent program should not depend on accidents of timing** .

To achieve that correctness, we enumerated four strategies for making code safe for concurrency (./20-thread-safety/) :

1. **Confinement** (./20-thread-safety/#strategy_1_confinement) : don't share data between threads.
2. **Immutability** (./20-thread-safety/#strategy_2_immutability) : make the shared data immutable.
3. **Use existing threadsafe data types** (./20-thread-safety/#strategy_3_using_threadsafe_data_types) : use a data type that does the coordination for you.
4. **Synchronization** : prevent threads from accessing the shared data at the same time. This is what we use to implement a threadsafe type, but we didn't discuss it at the time.

We talked about strategies 1-3 earlier. In this reading, we'll finish talking about strategy 4, using **synchronization** to implement your own data type that is **safe for shared-memory concurrency**.

Synchronization

The correctness of a concurrent program should not depend on accidents of timing.

Since race conditions caused by concurrent manipulation of shared mutable data are disastrous bugs — hard to discover, hard to reproduce, hard to debug — we need a way for concurrent modules that share memory to **synchronize** with each other.

Locks are one synchronization technique. A lock is an abstraction that allows at most one thread to *own* it at a time. *Holding a lock* is how one thread tells other threads: “I’m changing this thing, don’t touch it right now.”

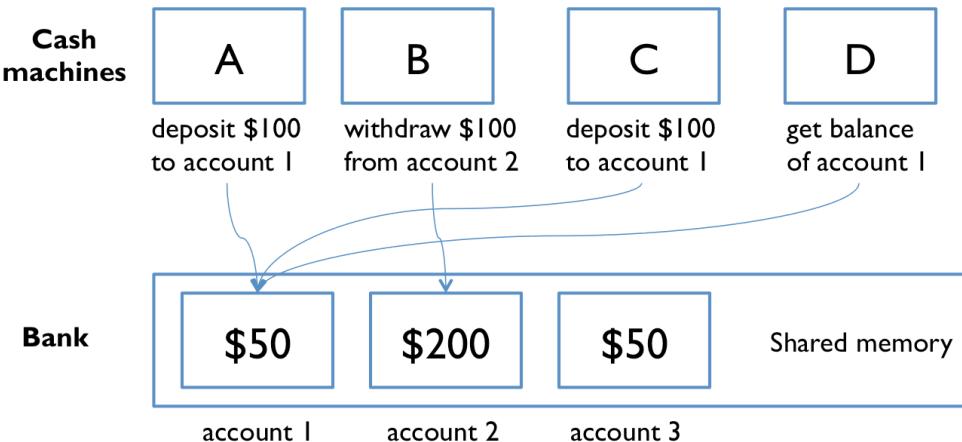
Locks have two operations:

- **acquire** allows a thread to take ownership of a lock. If a thread tries to acquire a lock currently owned by another thread, it *blocks* until the other thread releases the lock. At that point, it will contend with any other threads that are trying to acquire the lock. At most one thread can own the lock at a time.
- **release** relinquishes ownership of the lock, allowing another thread to take ownership of it.

Using a lock also tells the compiler and processor that you’re using shared memory concurrently, so that registers and caches will be flushed out to shared storage. This avoids the problem of reordering (./19-concurrency/#reordering), ensuring that the owner of a lock is always looking at up-to-date data.

Bank account example

Our first example of shared memory concurrency was a bank with cash machines (./19-)



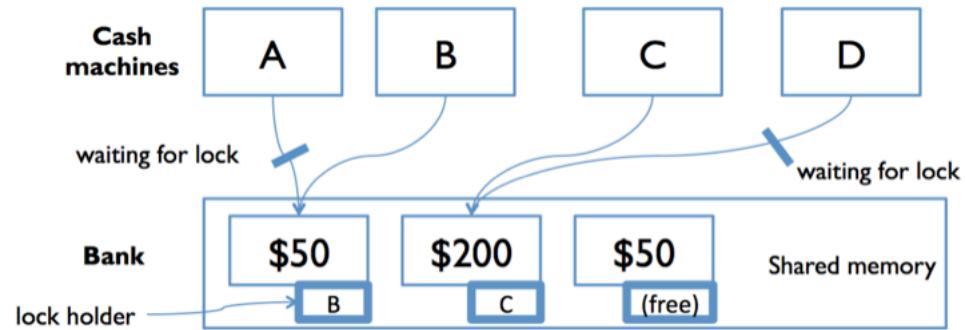
concurrency/#shared_memory_example). The diagram from that example is on the right.

The bank has several cash machines, all of which can read and write the same account objects in memory.

Of course, without any coordination between concurrent reads and writes to the account balances, things went horribly wrong (./19-concurrency/#interleaving) .

To solve this problem with locks, we can add a lock that protects each bank account. Now, before they can access or update an account balance, cash machines must first acquire the lock on that account.

In the diagram to the right, both A and B are trying to access account 1. Suppose B acquires the lock first. Then A must wait to read and write the balance until B finishes and releases the lock. This ensures that A and B are synchronized, but another cash machine C is able to run independently on a different account (because that account is protected by a different lock).

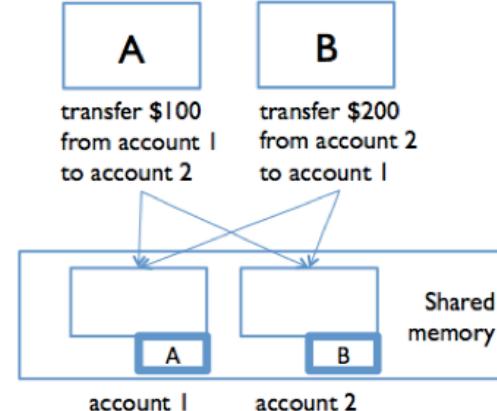


Deadlock

When used properly and carefully, locks can prevent race conditions. But then another problem rears its ugly head. Because the use of locks requires threads to wait (`acquire` blocks when another thread is holding the lock), it's possible to get into a situation where two threads are waiting *for each other* — and hence neither can make progress.

In the figure to the right, suppose A and B are making simultaneous transfers between two accounts in our bank.

A transfer between accounts needs to lock both accounts, so that money can't disappear from the system. A and B each acquire the lock on their respective "from" account: A acquires the lock on account 1, and B acquires the lock on account 2. Now, each must acquire the lock on their "to" account: so A is waiting for B to release the account 2 lock, and B is waiting for A to release the account 1 lock. Stalemate! A and B are frozen in a "deadly embrace," and accounts are locked up.



Deadlock occurs when concurrent modules are stuck waiting for each other to do something. A deadlock may involve more than two modules: the signal feature of deadlock is a **cycle of dependencies**, e.g. A is waiting for B which is waiting for C which is waiting for A. None of them can make progress.

You can also have deadlock without using any locks. For example, a message-passing system can experience deadlock when message buffers fill up. If a client fills up the server's buffer with requests, and then *blocks* waiting to add another request, the server may then fill up the client's buffer with results and then block itself. So the client is waiting for the server, and the server waiting for the client, and neither can make progress until the other one does. Again, deadlock ensues.

In the Java Tutorials, read:

- [Deadlock](https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html) ([//docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html](https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html)) (1 page)

Developing a threadsafe abstract data type

Let's see how to use synchronization to implement a threadsafe ADT.

You can see all the code for this example on GitHub: **edit buffer example** (<https://github.com/mit6005/sp16-ex23-editor>) . You are *not* expected to read and understand all the code. All the relevant parts are excerpted below.

Suppose we're building a multi-user editor, like Google Docs, that allows multiple people to connect to it and edit it at the same time. We'll need a mutable datatype to represent the text in the document. Here's the interface; basically it represents a string with insert and delete operations:

```
/* An EditBuffer represents a threadsafe mutable
 * string of characters in a text editor. */
editor/blob/master/src/editor/EditBuffer.java
public interface EditBuffer {

    /**
     * Modifies this by inserting a string.
     * @param pos position to insert at
     *             (requires 0 <= pos <= current buffer length)
     * @param ins string to insert
     */
    public void insert(int pos, String ins);

    /**
     * Modifies this by deleting a substring
     * @param pos starting position of substring to delete
     *             (requires 0 <= pos <= current buffer length)
     * @param len length of substring to delete
     *             (requires 0 <= len <= current buffer length - pos)
     */
    public void delete(int pos, int len);

    /**
     * @return length of text sequence in this edit buffer
     */
    public int length();

    /**
     * @return content of this edit buffer
     */
    public String toString();
}
```

A very simple rep for this datatype would just be a string:

```
SimpleBuffer.java (https://github.com/mit6005/sp16-ex23-editor/blob/master/src/editor/SimpleBuffer.java)
// Rep invariant:
//   text != null
// Abstraction function:
//   represents the sequence text[0],...,text[text.length()-1]
```

The downside of this rep is that every time we do an insert or delete, we have to copy the entire string into a new string. That gets expensive. Another rep we could use would be a character array, with space at the end. That's fine if the user is just typing new text at the end of the document (we don't have to

copy anything), but if the user is typing at the beginning of the document, then we're copying the entire document with every keystroke.

A more interesting rep, which is used by many text editors in practice, is called a *gap buffer*. It's basically a character array with extra space in it, but instead of having all the extra space at the end, the extra space is a *gap* that can appear anywhere in the buffer. Whenever an insert or delete operation needs to be done, the datatype first moves the gap to the location of the operation, and then does the insert or delete. If the gap is already there, then nothing needs to be copied — an insert just consumes part of the gap, and a delete just enlarges the gap! Gap buffers are particularly well-suited to representing a string that is being edited by a user with a cursor, since inserts and deletes tend to be focused around the cursor, so the gap rarely moves.

```
/* GapBuffer.java is a non-threadsafe EditBuffer that is optimized
 * for editing with a cursor, which tends to make a sequence of
 * inserts and deletes at the same place in the buffer. */
public class GapBuffer implements EditBuffer {
    private char[] a;
    private int gapStart;
    private int gapLength;
    // Rep invariant:
    //   a != null
    //   0 <= gapStart <= a.length
    //   0 <= gapLength <= a.length - gapStart
    // Abstraction function:
    //   represents the sequence a[0],...,a[gapStart-1],
    //   a[gapStart+gapLength],...,a[length-1]
```

In a multiuser scenario, we'd want multiple gaps, one for each user's cursor, but we'll use a single gap for now.

Steps to develop the datatype

Recall our recipe for designing and implementing an ADT:

1. **Specify.** Define the operations (method signatures and specs). We did that in the `EditBuffer` interface.
2. **Test.** Develop test cases for the operations. See `EditBufferTest` in the provided code. The test suite includes a testing strategy based on partitioning the parameter space of the operations.
3. **Rep.** Choose a rep. We chose two of them for `EditBuffer`, and this is often a good idea:
 - a. **Implement a simple, brute-force rep first.** It's easier to write, you're more likely to get it right, and it will validate your test cases and your specification so you can fix problems in them before you move on to the harder implementation. This is why we implemented `SimpleBuffer` before moving on to `GapBuffer`. Don't throw away your simple version, either — keep it around so that you have something to test and compare against in case things go wrong with the more complex one.
 - b. **Write down the rep invariant and abstraction function, and implement `checkRep()`.** `checkRep()` asserts the rep invariant at the end of every constructor, producer, and mutator method. (It's typically not necessary to call it at the end of an observer, since the rep hasn't changed.) In fact, assertions can be very useful for testing complex implementations, so it's not a bad idea to also assert the postcondition at the end of a complex method. You'll see an example of this in `GapBuffer.moveGap()` in the code with this reading.

In all these steps, we're working entirely single-threaded at first. Multithreaded clients should be in the back of our minds at all times while we're writing specs and choosing reps (we'll see later that careful choice of operations may be necessary to avoid race conditions in the clients of your datatype). But get it working, and thoroughly tested, in a sequential, single-threaded environment first.

Now we're ready for the next step:

4. **Synchronize.** Make an argument that your rep is threadsafe. Write it down explicitly as a comment in your class, right by the rep invariant, so that a maintainer knows how you designed thread safety into the class.

This part of the reading is about how to do step 4. We already saw how to make a thread safety argument ([./20-thread-safety/#how_to_make_a_safety_argument](#)) , but this time, we'll rely on synchronization in that argument.

And then the extra step we hinted at above:

5. **Iterate** . You may find that your choice of operations makes it hard to write a threadsafe type with the guarantees clients require. You might discover this in step 1, or in step 2 when you write tests, or in steps 3 or 4 when you implement. If that's the case, go back and refine the set of operations your ADT provides.

Locking

Locks are so commonly-used that Java provides them as a built-in language feature.

In Java, every object has a lock implicitly associated with it – a `String` , an array, an `ArrayList` , and every class you create, all of their object instances have a lock. Even a humble `Object` has a lock, so bare `Object` s are often used for explicit locking:

```
Object lock = new Object();
```

You can't call `acquire` and `release` on Java's intrinsic locks, however. Instead you use the **synchronized** statement to acquire the lock for the duration of a statement block:

```
synchronized (lock) { // thread blocks here until lock is free
    // now this thread has the lock
    balance = balance + 1;
    // exiting the block releases the lock
}
```

Synchronized regions like this provide **mutual exclusion** : only one thread at a time can be in a synchronized region guarded by a given object's lock. In other words, you are back in sequential programming world, with only one thread running at a time, at least with respect to other synchronized regions that refer to the same object.

Locks guard access to data

Locks are used to **guard** a shared data variable, like the account balance shown here. If all accesses to a data variable are guarded (surrounded by a synchronized block) by the same lock object, then those accesses will be guaranteed to be atomic — uninterrupted by other threads.

Because every object in Java has a lock implicitly associated with it, you might think that simply owning an object's lock would prevent other threads from accessing that object. **That is not the case.** Acquiring the lock associated with object `obj` using

```
synchronized (obj) { ... }
```

in thread t does one thing and one thing only: prevents other threads from entering a `synchronized(obj)` block, until thread t finishes its synchronized block. That's it.

Locks only provide mutual exclusion with other threads that acquire the same lock. All accesses to a data variable must be guarded by the same lock. You might guard an entire collection of variables behind a single lock, but all modules must agree on which lock they will all acquire and release.

Monitor pattern

When you are writing methods of a class, the most convenient lock is the object instance itself, i.e. `this`. As a simple approach, we can guard the entire rep of a class by wrapping all accesses to the rep inside `synchronized (this)`.

```
/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {
    private String text;
    ...
    public SimpleBuffer() {
        synchronized (this) {
            text = "";
            checkRep();
        }
    }
    public void insert(int pos, String ins) {
        synchronized (this) {
            text = text.substring(0, pos) + ins + text.substring(pos);
            checkRep();
        }
    }
    public void delete(int pos, int len) {
        synchronized (this) {
            text = text.substring(0, pos) + text.substring(pos+len);
            checkRep();
        }
    }
    public int length() {
        synchronized (this) {
            return text.length();
        }
    }
    public String toString() {
        synchronized (this) {
            return text;
        }
    }
}
```

Note the very careful discipline here. *Every* method that touches the rep must be guarded with the lock — even apparently small and trivial ones like `length()` and `toString()`. This is because reads must be guarded as well as writes — if reads are left unguarded, then they may be able to see the rep in a partially-modified state.

This approach is called the **monitor pattern**. A monitor is a class whose methods are mutually exclusive, so that only one thread can be inside an instance of the class at a time.

Java provides some syntactic sugar for the monitor pattern. If you add the keyword `synchronized` to a method signature, then Java will act as if you wrote `synchronized (this)` around the method body. So the code below is an equivalent way to implement the synchronized `SimpleBuffer` :

```
/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {
    private String text;
    ...
    public SimpleBuffer() {
        text = "";
        checkRep();
    }
    public synchronized void insert(int pos, String ins) {
        text = text.substring(0, pos) + ins + text.substring(pos);
        checkRep();
    }
    public synchronized void delete(int pos, int len) {
        text = text.substring(0, pos) + text.substring(pos+len);
        checkRep();
    }
    public synchronized int length() {
        return text.length();
    }
    public synchronized String toString() {
        return text;
    }
}
```

Notice that the `SimpleBuffer` constructor doesn't have a `synchronized` keyword. Java actually forbids it, syntactically, because an object under construction is expected to be confined to a single thread until it has returned from its constructor. So synchronizing constructors should be unnecessary.

In the Java Tutorials, read:

- **Synchronized Methods**
[\(1 page\)](https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html)
- **Intrinsic Locks and Synchronization**
[\(1 page\)](https://docs.oracle.com/javase/tutorial/essential/concurrency/locksSync.html)

Thread safety argument with synchronization

Now that we're protecting `SimpleBuffer`'s rep with a lock, we can write a better thread safety argument:

```
/** SimpleBuffer is a threadsafe EditBuffer with a simple rep. */
public class SimpleBuffer implements EditBuffer {
    private String text;
    // Rep invariant:
    //   text != null
    // Abstraction function:
    //   represents the sequence text[0],...,text[text.length()-1]
    // Thread safety argument:
    //   all accesses to text happen within SimpleBuffer methods,
    //   which are all guarded by SimpleBuffer's lock
```

The same argument works for `GapBuffer`, if we use the monitor pattern to synchronize all its methods.

Note that the encapsulation of the class, the absence of rep exposure, is very important for making this argument. If `text` were public:

```
public String text;
```

then clients outside `SimpleBuffer` would be able to read and write it without knowing that they should first acquire the lock, and `SimpleBuffer` would no longer be threadsafe.

Locking discipline

A locking discipline is a strategy for ensuring that synchronized code is threadsafe. We must satisfy two conditions:

1. Every shared mutable variable must be guarded by some lock. The data may not be read or written except inside a synchronized block that acquires that lock.
2. If an invariant involves multiple shared mutable variables (which might even be in different objects), then all the variables involved must be guarded by the *same* lock. Once a thread acquires the lock, the invariant must be reestablished before releasing the lock.

The monitor pattern as used here satisfies both rules. All the shared mutable data in the rep — which the rep invariant depends on — are guarded by the same lock.

Atomic operations

Consider a find-and-replace operation on the `EditBuffer` datatype:

```
/** Modifies buf by replacing the first occurrence of s with t.
 * If s not found in buf, then has no effect.
 * @returns true if and only if a replacement was made
 */
public static boolean findReplace(EditBuffer buf, String s, String t) {
    int i = buf.toString().indexOf(s);
    if (i == -1) {
        return false;
    }
    buf.delete(i, s.length());
    buf.insert(i, t);
    return true;
}
```

This method makes three different calls to `buf` — to convert it to a string in order to search for `s`, to delete the old text, and then to insert `t` in its place. Even though each of these calls individually is atomic, the `findReplace` method as a whole is not threadsafe, because other threads might mutate the buffer while `findReplace` is working, causing it to delete the wrong region or put the replacement back in the wrong place.

To prevent this, `findReplace` needs to synchronize with all other clients of `buf`.

Giving clients access to a lock

It's sometimes useful to make your datatype's lock available to clients, so that they can use it to implement higher-level atomic operations using your datatype.

So one approach to the problem with `findReplace` is to document that clients can use the `EditBuffer`'s lock to synchronize with each other:

```
/** An EditBuffer represents a threadsafe mutable string of characters
 * in a text editor. Clients may synchronize with each other using the
 * EditBuffer object itself. */
public interface EditBuffer {
    ...
}
```

And then `findReplace` can synchronize on `buf`:

```
public static boolean findReplace(EditBuffer buf, String s, String t) {
    synchronized (buf) {
        int i = buf.toString().indexOf(s);
        if (i == -1) {
            return false;
        }
        buf.delete(i, s.length());
        buf.insert(i, t);
        return true;
    }
}
```

The effect of this is to enlarge the synchronization region that the monitor pattern already put around the individual `toString`, `delete`, and `insert` methods, into a single atomic region that ensures that all three methods are executed without interference from other threads.

Sprinkling `synchronized` everywhere?

So is thread safety simply a matter of putting the `synchronized` keyword on every method in your program? Unfortunately not.

First, you actually don't want to synchronize methods willy-nilly. Synchronization imposes a large cost on your program. Making a synchronized method call may take significantly longer, because of the need to acquire a lock (and flush caches and communicate with other processors). Java leaves many of its mutable datatypes unsynchronized by default exactly for these performance reasons. When you don't need synchronization, don't use it.

Another argument for using `synchronized` in a more deliberate way is that it minimizes the scope of access to your lock. Adding `synchronized` to every method means that your lock is the object itself, and every client with a reference to your object automatically has a reference to your lock, that it can

acquire and release at will. Your thread safety mechanism is therefore public and can be interfered with by clients. Contrast that with using a lock that is an object internal to your rep, and acquired appropriately and sparingly using `synchronized()` blocks.

Finally, it's not actually sufficient to sprinkle `synchronized` everywhere. Dropping `synchronized` onto a method without thinking means that you're acquiring a lock without thinking about which lock it is, or about whether it's the right lock for guarding the shared data access you're about to do. Suppose we had tried to solve `findReplace`'s synchronization problem simply by dropping `synchronized` onto its declaration:

```
public static synchronized boolean findReplace(EditBuffer buf, ...) {
```

This wouldn't do what we want. It would indeed acquire a lock — because `findReplace` is a static method, it would acquire a static lock for the whole class that `findReplace` happens to be in, rather than an instance object lock. As a result, only one thread could call `findReplace` at a time — even if other threads want to operate on *different* buffers, which should be safe, they'd still be blocked until the single lock was free. So we'd suffer a significant loss in performance, because only one user of our massive multiuser editor would be allowed to do a find-and-replace at a time, even if they're all editing different documents.

Worse, however, it wouldn't provide useful protection, because other code that touches the document probably wouldn't be acquiring the same lock. It wouldn't actually eliminate our race conditions.

The `synchronized` keyword is not a panacea. Thread safety requires a discipline — using confinement, immutability, or locks to protect shared data. And that discipline needs to be written down, or maintainers won't know what it is.

Designing a datatype for concurrency

`findReplace`'s problem can be interpreted another way: that the `EditBuffer` interface really isn't that friendly to multiple simultaneous clients. It relies on integer indexes to specify insert and delete locations, which are extremely brittle to other mutations. If somebody else inserts or deletes before the index position, then the index becomes invalid.

So if we're designing a datatype specifically for use in a concurrent system, we need to think about providing operations that have better-defined semantics when they are interleaved. For example, it might be better to pair `EditBuffer` with a `Position` datatype representing a cursor position in the buffer, or even a `Selection` datatype representing a selected range. Once obtained, a `Position` could hold its location in the text against the wash of insertions and deletions around it, until the client was ready to use that `Position`. If some other thread deleted all the text around the `Position`, then the `Position` would be able to inform a subsequent client about what had happened (perhaps with an exception), and allow the client to decide what to do. These kinds of considerations come into play when designing a datatype for concurrency.

As another example, consider the `ConcurrentMap` ([//docs.oracle.com/javase/8/docs/api/?java/util/concurrent/ConcurrentMap.html](https://docs.oracle.com/javase/8/docs/api/?java/util/concurrent/ConcurrentMap.html)) interface in Java. This interface extends the existing `Map` interface, adding a few key methods that are commonly needed as atomic operations on a shared mutable map, e.g.:

- `map.putIfAbsent(key, value)` is an atomic version of
`if (! map.containsKey(key)) map.put(key, value);`
- `map.replace(key, value)` is an atomic version of
`if (map.containsKey(key)) map.put(key, value);`

Deadlock rears its ugly head

The locking approach to thread safety is powerful, but (unlike confinement and immutability) it introduces blocking into the program. Threads must sometimes wait for other threads to get out of synchronized regions before they can proceed. And blocking raises the possibility of deadlock — a very real risk, and frankly *far* more common in this setting than in message passing with blocking I/O (where we first mentioned it).

With locking, deadlock happens when threads acquire multiple locks at the same time, and two threads end up blocked while holding locks that they are each waiting for the other to release. The monitor pattern unfortunately makes this fairly easy to do. Here's an example.

Suppose we're modeling the social network of a series of books:

```
public class Wizard {
    private final String name;
    private final Set<Wizard> friends;
    // Rep invariant:
    //   name, friends != null
    //   friend links are bidirectional:
    //     for all f in friends, f.friends contains this
    // Concurrency argument:
    //   threadsafe by monitor pattern: all accesses to rep
    //   are guarded by this object's lock

    public Wizard(String name) {
        this.name = name;
        this.friends = new HashSet<Wizard>();
    }

    public synchronized boolean isFriendsWith(Wizard that) {
        return this.friends.contains(that);
    }

    public synchronized void friend(Wizard that) {
        if (friends.add(that)) {
            that.friend(this);
        }
    }

    public synchronized void defriend(Wizard that) {
        if (friends.remove(that)) {
            that.defriend(this);
        }
    }
}
```

Like Facebook, this social network is bidirectional: if x is friends with y , then y is friends with x . The `friend()` and `defriend()` methods enforce that invariant by modifying the reps of both objects, which because they use the monitor pattern means acquiring the locks to both objects as well.

Let's create a couple of wizards:

```
Wizard harry = new Wizard("Harry Potter");
Wizard snape = new Wizard("Severus Snape");
```

And then think about what happens when two independent threads are repeatedly running:

```
// thread A           // thread B
harry.friend(snape); snape.friend(harry);
harry.defriend(snape); snape.defriend(harry);
```

We will deadlock very rapidly. Here's why. Suppose thread A is about to execute `harry.friend(snape)`, and thread B is about to execute `snape.friend(harry)`.

- Thread A acquires the lock on `harry` (because the `friend` method is synchronized).
- Then thread B acquires the lock on `snape` (for the same reason).
- They both update their individual reps independently, and then try to call `friend()` on the other object — which requires them to acquire the lock on the other object.

So A is holding Harry and waiting for Snape, and B is holding Snape and waiting for Harry. Both threads are stuck in `friend()`, so neither one will ever manage to exit the synchronized region and release the lock to the other. This is a classic deadly embrace. The program simply stops.

The essence of the problem is acquiring multiple locks, and holding some of the locks while waiting for another lock to become free.

Notice that it is possible for thread A and thread B to interleave such that deadlock does not occur: perhaps thread A acquires and releases both locks before thread B has enough time to acquire the first one. If the locks involved in a deadlock are also involved in a race condition — and very often they are — then the deadlock will be just as difficult to reproduce or debug.

Deadlock solution 1: lock ordering

One way to prevent deadlock is to put an ordering on the locks that need to be acquired simultaneously, and ensuring that all code acquires the locks in that order.

In our social network example, we might always acquire the locks on the `Wizard` objects in alphabetical order by the wizard's name. Since thread A and thread B are both going to need the locks for Harry and Snape, they would both acquire them in that order: Harry's lock first, then Snape's. If thread A gets Harry's lock before B does, it will also get Snape's lock before B does, because B can't proceed until A releases Harry's lock again. The ordering on the locks forces an ordering on the threads acquiring them, so there's no way to produce a cycle in the waiting-for graph.

Here's what the code might look like:

```
public void friend(Wizard that) {
    Wizard first, second;
    if (this.name.compareTo(that.name) < 0) {
        first = this; second = that;
    } else {
        first = that; second = this;
    }
    synchronized (first) {
        synchronized (second) {
            if (friends.add(that)) {
                that.friend(this);
            }
        }
    }
}
```

(Note that the decision to order the locks alphabetically by the person's name would work fine for this book, but it wouldn't work in a real life social network. Why not? What would be better to use for lock ordering than the name?)

Although lock ordering is useful (particularly in code like operating system kernels), it has a number of drawbacks in practice.

- First, it's not modular — the code has to know about all the locks in the system, or at least in its subsystem.
- Second, it may be difficult or impossible for the code to know exactly which of those locks it will need before it even acquires the first one. It may need to do some computation to figure it out. Think about doing a depth-first search on the social network graph, for example — how would you know which nodes need to be locked, before you've even started looking for them?

Deadlock solution 2: coarse-grained locking

A more common approach than lock ordering, particularly for application programming (as opposed to operating system or device driver programming), is to use coarser locking — use a single lock to guard many object instances, or even a whole subsystem of a program.

For example, we might have a single lock for an entire social network, and have all the operations on any of its constituent parts synchronize on that lock. In the code below, all `Wizard`s belong to a `Castle`, and we just use that `Castle` object's lock to synchronize:

```
public class Wizard {
    private final Castle castle;
    private final String name;
    private final Set<Wizard> friends;
    ...
    public void friend(Wizard that) {
        synchronized (castle) {
            if (this.friends.add(that)) {
                that.friend(this);
            }
        }
    }
}
```

Coarse-grained locks can have a significant performance penalty. If you guard a large pile of mutable data with a single lock, then you're giving up the ability to access any of that data concurrently. In the worst case, having a single lock protecting everything, your program might be essentially sequential — only one thread is allowed to make progress at a time.

Goals of concurrent program design

Now is a good time to pop up a level and look at what we're doing. Recall that our primary goals are to create software that is **safe from bugs**, **easy to understand**, and **ready for change**.

Building concurrent software is clearly a challenge for all three of these goals. We can break the issues into two general classes. When we ask whether a concurrent program is *safe from bugs*, we care about two properties:

- **Safety.** Does the concurrent program satisfy its invariants and its specifications? Races in accessing mutable data threaten safety. Safety asks the question: can you prove that **some bad thing never happens**?

- **Liveness.** Does the program keep running and eventually do what you want, or does it get stuck somewhere waiting forever for events that will never happen? Can you prove that **some good thing eventually happens** ?

Deadlocks threaten liveness. Liveness may also require *fairness* , which means that concurrent modules are given processing capacity to make progress on their computations. Fairness is mostly a matter for the operating system's thread scheduler, but you can influence it (for good or for ill) by setting thread priorities.

Concurrency in practice

What strategies are typically followed in real programs?

- **Library data structures** either use no synchronization (to offer high performance to single-threaded clients, while leaving it to multithreaded clients to add locking on top) or the monitor pattern.
- **Mutable data structures with many parts** typically use either coarse-grained locking or thread confinement. Most graphical user interface toolkits follow one of these approaches, because a graphical user interface is basically a big mutable tree of mutable objects. Java Swing, the graphical user interface toolkit, uses thread confinement. Only a single dedicated thread is allowed to access Swing's tree. Other threads have to pass messages to that dedicated thread in order to access the tree.
- **Search** often uses immutable datatypes. Our Boolean formula satisfiability search (../16-recursive-data-types/recursive/#another_example_boolean_formulas) would be easy to make multithreaded, because all the datatypes involved were immutable. There would be no risk of either races or deadlocks.
- **Operating systems** often use fine-grained locks in order to get high performance, and use lock ordering to deal with deadlock problems.

We've omitted one important approach to mutable shared data because it's outside the scope of this course, but it's worth mentioning: **a database** . Database systems are widely used for distributed client/server systems like web applications. Databases avoid race conditions using *transactions* , which are similar to synchronized regions in that their effects are atomic, but they don't have to acquire locks, though a transaction may fail and be rolled back if it turns out that a race occurred. Databases can also manage locks, and handle locking order automatically. For more about how to use databases in system design, 6.170 Software Studio is strongly recommended; for more about how databases work on the inside, take 6.814 Database Systems.

And if you're interested in the **performance** of concurrent programs — since performance is often one of the reasons we add concurrency to a system in the first place — then 6.172 Performance Engineering is the course for you.

Summary

Producing a concurrent program that is safe from bugs, easy to understand, and ready for change requires careful thinking. Heisenbugs will skitter away as soon as you try to pin them down, so debugging simply isn't an effective way to achieve correct threadsafe code. And threads can interleave their operations in so many different ways that you will never be able to test even a small fraction of all possible executions.

- Make thread safety arguments about your datatypes, and document them in the code.

- Acquiring a lock allows a thread to have exclusive access to the data guarded by that lock, forcing other threads to block — as long as those threads are also trying to acquire that same lock.
- The *monitor pattern* guards the rep of a datatype with a single lock that is acquired by every method.
- Blocking caused by acquiring multiple locks creates the possibility of deadlock.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 24: Graphical User Interfaces

[View Tree](#)

[How the View Tree is Used](#)

[Input Handling](#)

[Separating Frontend from Backend](#)

[Background Processing in Graphical User Interfaces](#)

[Summary](#)

Reading 24: Graphical User Interfaces

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

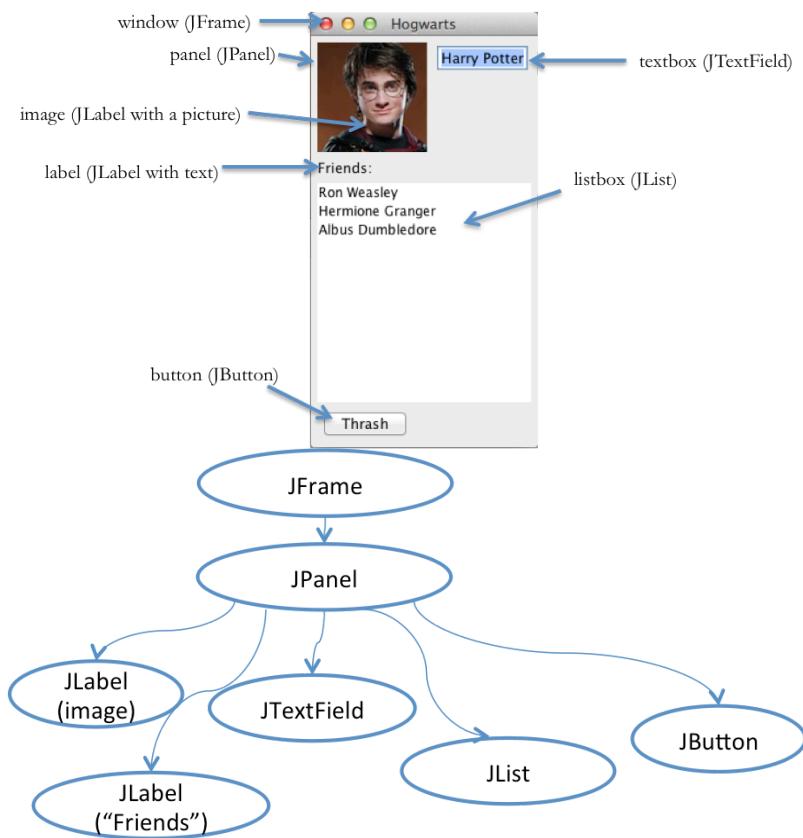
Today we'll take a high-level look at the software architecture of GUI software, focusing on the design patterns that have proven most useful. Three of the most important patterns are:

- the view tree, which is a central feature in the architecture of every important GUI toolkit;
- the model-view-controller pattern, which separates input, output, and data;
- the listener pattern, which is essential to decoupling the model from the view and controller.

View Tree

Graphical user interfaces are composed of view objects, each of which occupies a certain portion of the screen, generally a rectangular area called its bounding box. The view concept goes by a variety of names in various UI toolkits. In Java Swing ([https://en.wikipedia.org/wiki/Swing_\(Java\)](https://en.wikipedia.org/wiki/Swing_(Java))), they're `JComponent` objects; in HTML, they're elements or nodes; in other toolkits, they may be called widgets, controls, or interactors.

This leads to the first important pattern we'll talk about today: the view tree. Views are arranged into a hierarchy of containment, in which some views contain other views. Typical containers are windows, panels, and toolbars. The view tree is not just an arbitrary hierarchy, but is in fact a spatial one: child views are nested inside their parent's bounding box.



How the View Tree is Used

Virtually every GUI system has some kind of view tree. The view tree is a powerful structuring idea, which is loaded with responsibilities in a typical GUI:

Output. Views are responsible for displaying themselves, and the view tree directs the display process. GUIs change their output by mutating the view tree. For example, to show a new set of photos in a photo album GUI, the current thumbnails are removed from the view tree and a new set of thumbnails is added in their place. A redraw algorithm built into the GUI toolkit automatically redraws the affected parts of the subtree. In Java Swing, every view in the tree has a `paint()` method that knows how to draw itself on the screen. The repaint process is driven by calling `paint()` on the root of the tree, which recursively calls `paint()` down through all the descendent nodes of the view tree.

Input. Views can have input handlers, and the view tree controls how mouse and keyboard input is processed. More on this in a moment.

Layout. The view tree controls how the views are laid out on the screen, i.e. how their bounding boxes are assigned. An automatic layout algorithm automatically calculates positions and sizes of views. Specialized containers (like `JSplitPane`, `JScrollPane`) do layout themselves. More generic containers (`JPanel`, `JFrame`) delegate layout decisions to a layout manager (e.g. `GridLayout`, `BorderLayout`, `BoxLayout`, ...).

Input Handling

Input is handled somewhat differently in GUIs than we've been handling it in parsers and servers. In those systems, we've seen a single parser that peels apart the input and decides how to direct it to different modules of the program. If a GUI were written that way, it might look like this (in pseudocode):

```

while (true) {
    read mouse click
    if (clicked on Thrash button) doThrash();
    else if (clicked on textbox) doPlaceCursor();
    else if (clicked on a name in the listbox) doSelectItem();
    ...
}

```

In a GUI, we don't directly write this kind of method, because it's not modular – it mixes up responsibilities for button, listbox, and textbox all in one place. Instead, GUIs exploit the spatial separation provided by the view tree to provide functional separation as well. Mouse clicks and keyboard events are distributed around the view tree, depending on where they occur.

GUI input event handling is an instance of the **Listener pattern** (also known as Publish-Subscribe). In the Listener pattern:

- An event source generates a stream of discrete events, which correspond to state transitions in the source.
- One or more listeners register interest (subscribe) to the stream of events, providing a function to be called when a new event occurs.

In this case, the mouse is the event source, and the events are changes in the state of the mouse: its x,y position or the state of its buttons (whether they are pressed or released). Events often include additional information about the transition (such as the x,y position of mouse), which might be bundled into an event object or passed as parameters.

When an event occurs, the event source distributes it to all subscribed listeners, by calling their callback methods.

The control flow through a graphical user interface proceeds like this:

- A top-level event loop reads input from mouse and keyboard. In Java Swing, and most graphical user interface toolkits, this loop is actually hidden from you. It's buried inside the toolkit, and listeners appear to be called magically.
- For each input event, it finds the right view in the tree (by looking at the x,y position of the mouse) and sends the event to that view's listeners.
- Each listener does its thing (which might involve e.g. modifying objects in the view tree), and then *returns immediately to the event loop*.

The last part – listeners return to the event loop as fast as possible – is very important, because it preserves the responsiveness of the user interface. We'll come back to this later in the reading.

The Listener pattern isn't just used for low-level input events like mouse clicks and keyboard keypresses. Many GUI objects generate their own higher-level events, often as a result of some combination of low-level input events. For example:

- JButton sends an action event when it is pressed (whether by mouse or keyboard)
- JList sends a selection event when the selected element changes (whether by mouse or by keyboard)
- JTextField sends change events when the text inside it changes for any reason

A button can be pressed either by the mouse (with a mouse down and mouse up event) or by the keyboard (which is important for people who can't use a mouse, like blind users). So you should always listen for these high-level events, not the low-level input events. Use an ActionListener to respond to a JButton press, not a mouse listener.

Separating Frontend from Backend

We've seen how GUI programs are structured around a view tree, and how input events are handled by attaching listeners to views. This is the start of a separation of concerns – output handled by views, and input handled by listeners.

But we're still missing the application itself – the backend that represents the data and logic that the user interface is showing and editing. (Why do we want to separate this from the user interface?)

The **Model-View-Controller pattern** has this separation of concerns as its primary goal. It separates the user interface frontend from the application backend, by putting backend code into the model and frontend code into the view and controller. MVC also separates input from output; the controller is supposed to handle input, and the view is supposed to handle output.

The **model** is responsible for maintaining application-specific data and providing access to that data. Models are often mutable, and they provide methods for changing the state safely, preserving its representation invariants. OK, all mutable objects do that. But a model must also notify its clients when there are changes to its data, so that dependent views can update their displays, and dependent controllers can respond appropriately.

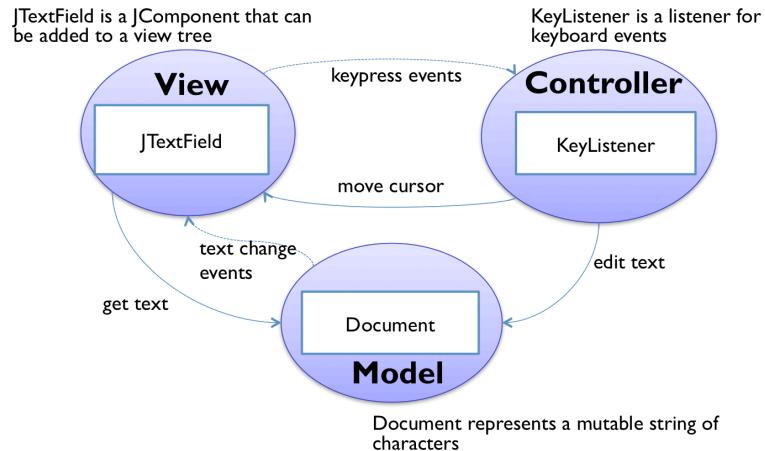
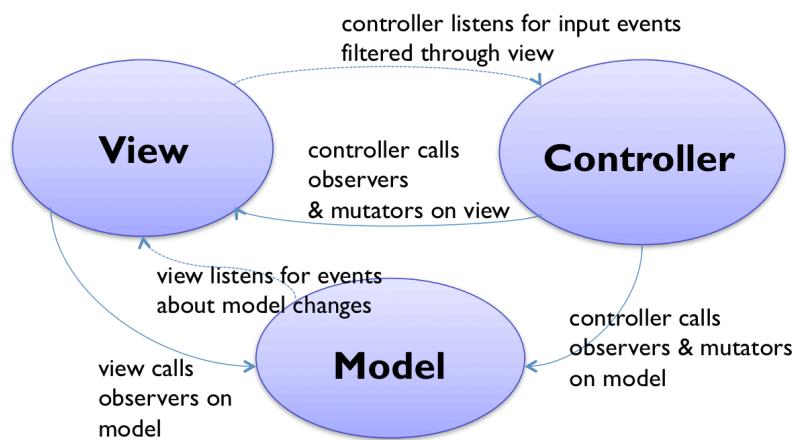
Models do this notification using the listener pattern, in which interested views and controllers register themselves as listeners for change events generated by the model.

View objects are responsible for output. A view usually occupies some chunk of the screen, usually a rectangular area. Basically, the view queries the model for data and draws the data on the screen. It listens for changes from the model so that it can update the screen to reflect those changes.

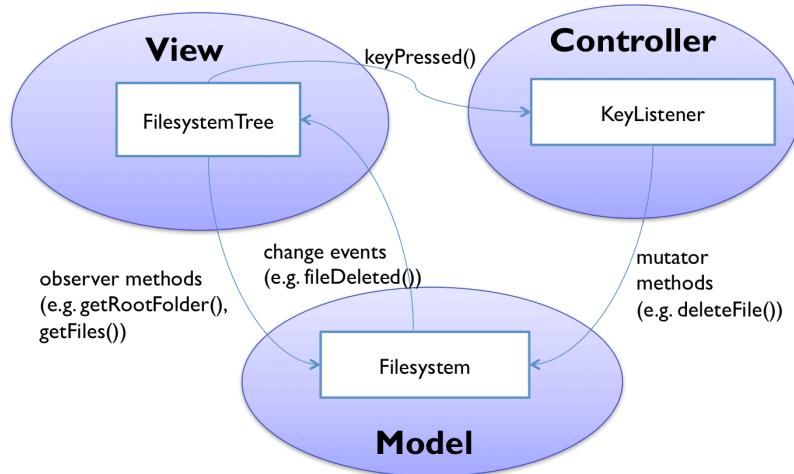
Finally, the **controller** handles the input. It receives keyboard and mouse events, and instructs the model to change accordingly.

A simple example of the MVC pattern is a text field. The figure at right shows Java Swing's text field, called `JTextField`. Its model is a mutable string of characters. The view is an object that draws the text on the screen (usually with a rectangle around it to indicate that it's an editable text field). The controller is an object that receives keystrokes typed by the user and inserts them into the mutable string.

Instances of the MVC pattern appear at many scales in GUI software. At a higher level, this text field might be part of a view (like an address book editor), with a different controller listening to it (for text-changed events), for a different model (like the address book). But when you drill down to a lower level, the text field itself is an instance of MVC.



Here's a larger example, in which the view is a filesystem browser (like the Mac Finder or Windows Explorer), the model is the disk filesystem, and the controller is an input handler that translates the user's keystrokes and mouse clicks into operations on the model and view.



The separation of model and view has several benefits. First, it allows the interface to have multiple views showing the same application data. For example, a database field might be shown in a table and in an editable form at the same time. Second, it allows views and models to be reused in other applications. The MVC pattern enables the creation of user interface toolkits, which are libraries of reusable views. Java Swing is such a toolkit. You can easily reuse view classes from this library (like JButton and JTree) while plugging your own models into them.

Background Processing in Graphical User Interfaces

The last major topic for today connects back to concurrency.

First, some motivation. Why do we need to do background processing in graphical user interfaces? Even though computer systems are steadily getting faster, we're also asking them to do more. Many programs need to do operations that may take some time: retrieving URLs over the network, running database queries, scanning a filesystem, doing complex calculations, etc.

But graphical user interfaces are event-driven programs, which means (generally speaking) everything is triggered by an input event handler. For example, in a web browser, clicking a hyperlink starts loading a new web page. But if the click handler is written so that it actually retrieves the web page itself, then the web browser will be very painful to use. Why? Because its interface will appear to freeze up until the click handler finishes retrieving the web page and returns to the event loop. Here's why.

This happens because input handling and screen repainting is all handled from a single thread. That thread (called the event-dispatch thread) has a loop that reads an input event from the queue and dispatches it to listeners on the view tree. When there are no input events left to process, it repaints the screen. But if an input handler you've written delays returning to this loop – because it's blocking on a network read, or because it's searching for the solution to a big Sudoku puzzle – then input events stop being handled, and the screen stops updating. So long tasks need to run in the background.

In Java, the event-dispatch thread is distinct from the main thread of the program (see below). It is started automatically when a user interface object is created. As a result, every Java GUI program is automatically multithreaded. Many programmers don't notice, because the main thread typically doesn't do much in a GUI program – it starts creation of the view, and then the main thread just exits, leaving only the event-dispatch thread to do the main work of the program.

The fact that Swing programs are multithreaded by default creates risks. There's very often a shared mutable datatype in your GUI: the model. If you use background threads to modify the model without blocking the event-dispatch thread, then you have to make sure your data structure is threadsafe.

But another important shared mutable datatype in your GUI is the view tree. Java Swing's view tree is not threadsafe. In general, you cannot safely call methods on a Swing object from anywhere but the event-dispatch thread.

The view tree is a big meatball of shared state, and the Swing specification doesn't guarantee that there's any lock protecting it. Instead the view tree is **confined to the event-dispatch thread**, by specification. So it's ok to access view objects from the event-dispatch thread (i.e., in response to input events), but the Swing specification forbids touching – reading or writing – any `JComponent` objects from a different thread. See Swing threading and the event-dispatch thread (<http://www.javaworld.com/javaworld/jw-08-2007/jw-08-swingthreading.html>) .

In the actual Swing implementation, there is one big lock (`Component.getTreeLock()` (<https://docs.oracle.com/javase/8/docs/api/java.awt/Component.html#getTreeLock-->)) but only some Swing methods use it, so it's not effective as a synchronization mechanism.

The safe way to access the view tree is to do it from the event-dispatch thread. So Swing takes a clever approach: it uses the event queue itself as a message-passing queue. In other words, you can put your own custom messages on the event queue, the same queue used for mouse clicks, keypresses, button action events, and so forth. Your custom message is actually a piece of executable code, an object that implements `Runnable` , and you put it on the queue using `SwingUtilities.invokeLater` (<https://docs.oracle.com/javase/8/docs/api/javax/swing/SwingUtilities.html#invokeLater-java.lang.Runnable->) . For example:

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        content.add(thumbnail);
        ...
    }
});
```

The `invokeLater()` drops this `Runnable` object at the end of the queue, and when Swing's event loop reaches it, it simply calls `run()` . Thus the body of `run()` ends up run by the event-dispatch thread, where it can safely call observers and mutators on the view tree.

In the Java Tutorials, read:

- **Concurrency in Swing**
(<https://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html>) (1 page)
- **Initial Threads** (<https://docs.oracle.com/javase/tutorial/uiswing/concurrency/initial.html>) (1 page)
- **The Event Dispatch Thread**
(<https://docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html>) (1 page)

Summary

- The view tree organizes the screen into a tree of nested rectangles, and it is used in dispatching input events as well as displaying output.
- The Listener pattern sends a stream of events (like mouse or keyboard events, or button action events) to registered listeners.
- The Model-View-Controller pattern separates responsibilities: model=data, view=output, controller=input.

- Long-running processing should be moved to a background thread, but the Swing view tree is confined to the event-dispatch thread. So accessing Swing objects from another thread requires using the event loop as a message-passing queue, to get back to the event-dispatch thread.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 25: Map, Filter, Reduce

Introduction: an example

Abstracting out control flow

Map

Functions as values

Filter

Reduce

Back to the intro example

Benefits of abstracting out control

First-class functions in Java

Map/filter/reduce in Java

Higher-order functions in Java

Summary

Reading 25: Map, Filter, Reduce

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

In this reading you'll learn a design pattern for implementing functions that operate on sequences of elements, and you'll see how treating functions themselves as *first-class values* that we can pass around and manipulate in our programs is an especially powerful idea.

- Map/filter/reduce
- Lambda expressions
- Functional objects
- Higher-order functions

Introduction: an example

Suppose we're given the following problem: write a method that finds the words in the Java files in your project.

Following good practice, we break it down into several simpler steps and write a method for each one:

- find all the files in the project, by scanning recursively from the project's root folder
- restrict them to files with a particular suffix, in this case `.java`
- open each file and read it in line-by-line
- break each line into words

Writing the individual methods for these substeps, we'll find ourselves writing a lot of low-level iteration code. For example, here's what the recursive traversal of the project folder might look like:

```
/** 
 * Find all the files in the filesystem subtree rooted at folder.
 * @param folder root of subtree, requires folder.isDirectory() == true
 * @return list of all ordinary files (not folders) that have folder as
 *         their ancestor
 */
public static List<File> allFilesIn(File folder) {
    List<File> files = new ArrayList<>();
    for (File f : folder.listFiles()) {
        if (f.isDirectory()) {
            files.addAll(allFilesIn(f));
        } else if (f.isFile()) {
            files.add(f);
        }
    }
    return files;
}
```

And here's what the filtering method might look like, which restricts that file list down to just the Java files (imagine calling this like `onlyFilesWithSuffix(files, ".java")`):

```
/** 
 * Filter a list of files to those that end with suffix.
 * @param files list of files (all non-null)
 * @param suffix string to test
 * @return a new list consisting of only those files whose names end with
 *         suffix
 */
public static List<File> onlyFilesWithSuffix(List<File> files, String suffix) {
    List<File> result = new ArrayList<>();
    for (File f : files) {
        if (f.getName().endsWith(suffix)) {
            result.add(f);
        }
    }
    return result;
}
```

→ **full Java code for the example** (<https://github.com/mit6005/sp16-ex25-words/blob/master/src/words/Words1.java>)

In this reading we discuss *map/filter/reduce*, a design pattern that substantially simplifies the implementation of functions that operate over sequences of elements. In this example, we'll have lots of sequences — lists of files; input streams that are sequences of lines; lines that are sequences of words; frequency tables that are sequences of (word, count) pairs. *Map/filter/reduce* will enable us to operate on those sequences with no explicit control flow — not a single `for` loop or `if` statement.

Along the way, we'll also see an important Big Idea: functions as "first-class" data values, meaning that they can be stored in variables, passed as arguments to functions, and created dynamically like other values.

Using first-class functions in Java is more verbose, uses some unfamiliar syntax, and the interaction with static typing adds some complexity. So to get started with map/filter/reduce, we'll switch back to Python.

Abstracting out control flow

We've already seen one design pattern that abstracts away from the details of iterating over a data structure: Iterator.

Iterator abstraction

Iterator gives you a sequence of elements from a data structure, without you having to worry about whether the data structure is a set or a token stream or a list or an array — the `Iterator` ([//docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html](https://docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html)) looks the same no matter what the data structure is.

For example, given a `List<File>` `files`, we can iterate using indices:

```
for (int ii = 0; ii < files.size(); ii++) {
    File f = files.get(ii);
    // ...
```

But this code depends on the `size` and `get` methods of `List`, which might be different in another data structure. Using an iterator abstracts away the details:

```
Iterator<File> iter = files.iterator();
while (iter.hasNext()) {
    File f = iter.next();
    // ...
```

Now the loop will be identical for any type that provides an `Iterator`. There is, in fact, an interface for such types: `Iterable` ([//docs.oracle.com/javase/8/docs/api/?java/lang/Iterable.html](https://docs.oracle.com/javase/8/docs/api/?java/lang/Iterable.html)). Any `Iterable` can be used with Java's enhanced for statement (<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html>) — `for (File f : files)` — and under the hood, it uses an iterator.

Map/filter/reduce abstraction

The map/filter/reduce patterns in this reading do something similar to Iterator, but at an even higher level: they treat the entire sequence of elements as a unit, so that the programmer doesn't have to name and work with the elements individually. In this paradigm, the control statements disappear: specifically, the `for` statements, the `if` statements, and the `return` statements in the code from our introductory example will be gone. We'll also be able to get rid of most of the temporary names (i.e., the local variables `files`, `f`, and `result`).

Sequences

Let's imagine an abstract datatype `Seq<E>` that represents a *sequence* of elements of type `E`.

For example, `[1, 2, 3, 4] ∈ Seq<Integer>`.

Any datatype that has an iterator can qualify as a sequence: array, list, set, etc. A string is also a sequence (of characters), although Java's strings don't offer an iterator. Python is more consistent in this respect: not only are lists iterable, but so are strings, tuples (which are immutable lists), and even input streams (which produce a sequence of lines). We'll see these examples in Python first, since the syntax is very readable and familiar to you, and then we'll see how it works in Java.

We'll have three operations for sequences: map, filter, and reduce. Let's look at each one in turn, and then look at how they work together.

Map

Map applies a unary function to each element in the sequence and returns a new sequence containing the results, in the same order:

map : ($E \rightarrow F$) \times Seq $<E>$ \rightarrow Seq $<F>$

For example, in Python:

```
>>> from math import sqrt
>>> map(sqrt, [1, 4, 9, 16])
[1.0, 2.0, 3.0, 4.0]
>>> map(str.lower, ['A', 'b', 'C'])
['a', 'b', 'c']
```

`map` is built-in, but it is also straightforward to implement in Python:

```
def map(f, seq):
    result = []
    for elt in seq:
        result.append(f(elt))
    return result
```

This operation captures a common pattern for operating over sequences: doing the same thing to each element of the sequence.

Functions as values

Let's pause here for a second, because we're doing something unusual with functions. The `map` function takes a reference to a *function* as its first argument — not to the result of that function. When we wrote

```
map(sqrt, [1, 4, 9, 16])
```

we didn't *call* `sqrt` (like `sqrt(25)` is a call), instead we just used its name. In Python, the name of a function is a reference to an object representing that function. You can assign that object to another variable if you like, and it still behaves like `sqrt` :

```
>>> mySquareRoot = sqrt
>>> mySquareRoot(25)
5.0
```

You can also pass a reference to the function object as a parameter to another function, and that's what we're doing here with `map`. You can use function objects the same way you would use any other data value in Python (like numbers or strings or objects).

Functions are **first-class** in Python, meaning that they can be assigned to variables, passed as parameters, used as return values, and stored in data structures. First-class functions are a very powerful programming idea. The first practical programming language that used them was Lisp, invented by John McCarthy at MIT. But the idea of programming with functions as first-class values actually predates computers, tracing back to Alonzo Church's lambda calculus. The lambda calculus used the Greek letter λ to define new functions; this term stuck, and you'll see it as a keyword not only in Lisp and its descendants, but also in Python.

We've seen how to use built-in library functions as first-class values; how do we make our own? One way is using a familiar function definition, which gives the function a name:

```
>>> def powerOfTwo(k):
...     return 2**k
...
>>> powerOfTwo(5)
32
>>> map(powerOfTwo, [1, 2, 3, 4])
[2, 4, 8, 16]
```

When you only need the function in one place, however — which often comes up in programming with functions — it's more convenient to use a **lambda expression** :

```
lambda k: 2**k
```

This expression represents a function of one argument (called `k`) that returns the value 2^k . You can use it anywhere you would have used `powerOfTwo` :

```
>>> (lambda k: 2**k)(5)
32
>>> map(lambda k: 2**k, [1, 2, 3, 4])
[2, 4, 8, 16]
```

Python lambda expressions are unfortunately syntactically limited, to functions that can be written with just a `return` statement and nothing else (no `if` statements, no `for` loops, no local variables). But remember that's our goal with map/filter/reduce anyway, so it won't be a serious obstacle.

Guido Von Rossum, the creator of Python, wrote a blog post about the design principle that led not only to first-class functions in Python, but first-class methods as well: First-class Everything ([//python-history.blogspot.com/2009/02/first-class-everything.html](http://python-history.blogspot.com/2009/02/first-class-everything.html)) .

More ways to use map

Map is useful even if you don't care about the return value of the function. When you have a sequence of mutable objects, for example, you can map a mutator operation over them:

```
map(IOBase.close, streams) # closes each stream on the list
map(Thread.join, threads) # waits for each thread to finish
```

Some versions of map (including Python's built-in `map`) also support mapping functions with multiple arguments. For example, you can add two lists of numbers element-wise:

```
>>> import operator
>>> map(operator.add, [1, 2, 3], [4, 5, 6])
[5, 7, 9]
```

Filter

Our next important sequence operation is **filter**, which tests each element with a unary predicate. Elements that satisfy the predicate are kept; those that don't are removed. A new list is returned; filter doesn't modify its input list.

filter : (E → boolean) × Seq<E> → Seq<E>

Python examples:

```
>>> filter(str.isalpha, ['x', 'y', '2', '3', 'a'])
['x', 'y', 'a']
```

```
>>> def isOdd(x): return x % 2 == 1
...
>>> filter(isOdd, [1, 2, 3, 4])
[1, 3]
```

```
>>> filter(lambda s: len(s)>0, ['abc', '', 'd'])
['abc', 'd']
```

We can define filter in a straightforward way:

```
def filter(f, seq):
    result = []
    for elt in seq:
        if f(elt):
            result.append(elt)
    return result
```

Reduce

Our final operator, **reduce**, combines the elements of the sequence together, using a binary function. In addition to the function and the list, it also takes an *initial value* that initializes the reduction, and that ends up being the return value if the list is empty.

reduce : (F × E → F) × Seq<E> × F → F

`reduce(f, list, init)` combines the elements of the list from left to right, as follows:

```
result_0 = init
result_1 = f(result_0, list[0])
result_2 = f(result_1, list[1])
...
result_n = f(result_{n-1}, list[n-1])
```

`result_n` is the final result for an n -element list.

Adding numbers is probably the most straightforward example:

```
>>> reduce(lambda x,y: x+y, [1, 2, 3], 0)
6
# --or--
>>> import operator
>>> reduce(operator.add, [1, 2, 3], 0)
6
```

There are two design choices in the reduce operation. First is whether to require an initial value. In Python's reduce function, the initial value is optional, and if you omit it, reduce uses the first element of the list as its initial value. So you get behavior like this instead:

```
result_0 = undefined (reduce throws an exception if the list is empty)
result_1 = list[0]
result_2 = f(result_1, list[1])
...
result_n = f(result_{n-1}, list[n-1])
```

This makes it easier to use reducers like `max`, which have no well-defined initial value:

```
>>> reduce(max, [5, 8, 3, 1])
8
```

The second design choice is the order in which the elements are accumulated. For associative operators like `add` and `max` it makes no difference, but for other operators it can. Python's reduce is also called **fold-left** in other programming languages, because it combines the sequence starting from the left (the first element). **Fold-right** goes in the other direction:

fold-right : (E × F → F) × Seq<E> × F → F

where `fold-right(f, list, init)` of an n-element list follows this pattern:

```
result_0 = init
result_1 = f(list[n-1], result_0)
result_2 = f(list[n-2], result_1)
...
result_n = f(list[0], result_{n-1})
```

to produce `result_n` as the final result.

Here's a diagram of two ways to reduce: from the left or from the right:

fold-left : (F × E → F) × Seq<E> × F → F

fold-left(-, [1, 2, 3], 0) = -6



fold-right : (E × F → F) × Seq<E> × F → F

fold-right(-, [1, 2, 3], 0) = 2

The return type of the reduce operation doesn't have to match the type of the list elements. For example, we can use reduce to glue together a sequence into a string:

```
>>> reduce(lambda s,x: s+str(x), [1, 2, 3, 4], '')  
'1234'
```

Or to flatten out nested sublists into a single list:

```
>>> reduce(operator.concat, [[1, 2], [3, 4], [], [5]], [])  
[1, 2, 3, 4, 5]
```

This is a useful enough sequence operation that we'll define it as **flatten**, although it's just a reduce step inside:

```
def flatten(list):  
    return reduce(operator.concat, list, [])
```

More examples

Suppose we have a polynomial represented as a list of coefficients, $a[0], a[1], \dots, a[n-1]$, where $a[i]$ is the coefficient of x^i . Then we can evaluate it using map and reduce:

```
def evaluate(a, x):  
    xi = map(lambda i: x**i, range(0, len(a))) # [x^0, x^1, x^2, ..., x^(n-1)]  
    axi = map(operator.mul, a, xi)           # [a[0]*x^0, a[1]*x^1, ..., a[n-1]  
    *x^(n-1)]  
    return reduce(operator.add, axi, 0)       # sum of axi
```

This code uses the convenient Python generator method `range(a,b)`, which generates a list of integers from a to b-1. In map/filter/reduce programming, this kind of method replaces a `for` loop that indexes from a to b.

Now let's look at a typical database query example. Suppose we have a database about digital cameras, in which each object is of type `Camera` with observer methods for its properties (`brand()`, `pixels()`, `cost()`, etc.). The whole database is in a list called `cameras`. Then we can describe queries on this database using map/filter/reduce:

```
# What's the highest resolution Nikon sells?  
reduce(max, map(Camera.pixels, filter(lambda c: c.brand() == "Nikon", cameras)))
```

Relational databases use the map/filter/reduce paradigm (where it's called project/select/aggregate). SQL (<https://en.wikipedia.org/wiki/SQL>) (Structured Query Language) is the *de facto* standard language for querying relational databases. A typical SQL query looks like this:

```
select max(pixels) from cameras where brand = "Nikon"
```

`cameras` is a **sequence** (a list of rows, where each row has the data for one camera)

`where brand = "Nikon"` is a **filter**

`pixels` is a **map** (extracting just the pixels field from the row)

`max` is a **reduce**

Back to the intro example

Going back to the example we started with, where we want to find all the words in the Java files in our project, let's try creating a useful abstraction for filtering files by suffix:

```
def fileEndsWith(suffix):
    return lambda file: file.getName().endsWith(suffix)
```

`fileEndsWith` returns *functions* that are useful as filters: it takes a filename suffix like `.java` and dynamically generates a function that we can use with `filter` to test for that suffix:

```
filter(fileEndsWith(".java"), files)
```

`fileEndsWith` is a different kind of beast than our usual functions. It's a **higher-order function**, meaning that it's a function that takes another function as an argument, or returns another function as its result. Higher-order functions are operations on the datatype of functions; in this case, `fileEndsWith` is a *producer* of functions.

Now let's use `map`, `filter`, and `flatten` (which we defined above using `reduce`) to recursively traverse the folder tree:

```
def allFilesIn(folder):
    children = folder.listFiles()
    subfolders = filter(File.isDirectory, children)
    descendants = flatten(map(allFilesIn, subfolders))
    return descendants + filter(File.isFile, children)
```

The first line gets all the children of the folder, which might look like this:

```
["src/client", "src/server", "src/Main.java", ...]
```

The second line is the key bit: it filters the children for just the subfolders, and then recursively maps `allFilesIn` against this list of subfolders! The result might look like this:

```
[["src/client/MyClient.java", ...], ["src/server/MyServer.java", ...], ...]
```

So we have to flatten it to remove the nested structure. Then we add the immediate children that are plain files (not folders), and that's our result.

We can also do the other pieces of the problem with `map/filter/reduce`. Once we have the list of files we want to extract words from, we're ready to load their contents. We can use `map` to get their pathnames as strings, open them, and then read in each file as a list of files:

```
pathnames = map(File.getPath, files)
streams = map(open, pathnames)
lines = map(list, streams)
```

This actually looks like a single `map` operation where we want to apply three functions to the elements, so let's pause to create another useful higher-order function: composing functions together.

```
def compose(f, g):
    """Requires that f and g are functions, f:A->B and g:B->C.
    Returns a function A->C by composing f with g."""
    return lambda x: g(f(x))
```

Now we can use a single map:

```
lines = map(compose(compose(File.getPath, open), list), files)
```

Better, since we already have three functions to apply, let's design a way to compose an arbitrary chain of functions:

```
def chain(funcs):
    """Requires funcs is a list of functions [A->B, B->C, ..., Y->Z].
    Returns a fn A->Z that is the left-to-right composition of funcs."""
    return reduce(compose, funcs)
```

So that the map operation becomes:

```
lines = map(chain([File.getPath, open, list]), files)
```

Now we see more of the power of first-class functions. We can put functions into data structures and use operations on those data structures, like map, reduce, and filter, on the functions themselves!

Since this map will produce a list of lists of lines (one list of lines for each file), let's flatten it to get a single line list, ignoring file boundaries:

```
allLines = flatten(map(chain([File.getPath, open, list]), files))
```

Then we split each line into words similarly:

```
words = flatten(map(str.split, lines))
```

And we're done, we have our list of all words in the project's Java files! As promised, the control statements have disappeared.

→ **full Python code for the example** (<https://github.com/mit6005/sp16-ex25-words/blob/master/src/words/Words2.py>)

Benefits of abstracting out control

Map/filter/reduce can often make code shorter and simpler, and allow the programmer to focus on the heart of the computation rather than on the details of loops, branches, and control flow.

By arranging our program in terms of map, filter, and reduce, and in particular using immutable datatypes and pure functions (functions that do not mutate data) as much as possible, we've created more opportunities for safe concurrency. Maps and filters using pure functions over immutable datatypes are instantly parallelizable — invocations of the function on different elements of the sequence can be run in different threads, on different processors, even on different machines, and the result will still be the same. MapReduce (<https://en.wikipedia.org/wiki/MapReduce>) is a pattern for parallelizing large computations in this way.

First-class functions in Java

We've seen what first-class functions look like in Python; how does this all work in Java?

In Java, the only first-class values are primitive values (ints, booleans, characters, etc.) and object references. But objects can carry functions with them, in the form of methods. So it turns out that the way to implement a first-class function, in an object-oriented programming language like Java that doesn't support first-class functions directly, is to use an object with a method representing the function.

We've actually seen this before several times already:

- The `Runnable` object that you pass to a `Thread` constructor is a first-class function, `void run()`.
- The `Comparator<T>` object that you pass to a sorted collection (e.g. `SortedSet`) is a first-class function, `int compare(T o1, T o2)`.
- The `KeyListener` object that you register with the graphical user interface toolkit to get keyboard events is a bundle of several functions, `keyPressed(KeyEvent)`, `keyReleased(KeyEvent)`, etc.

This design pattern is called a **functional object** or **functor**, an object whose purpose is to represent a function.

Lambda expressions in Java

Java's lambda expression syntax provides a succinct way to create instances of functional objects. For example, instead of writing:

```
new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello!");
    }
}).start();
```

we can use a lambda expression:

```
new Thread(() -> {
    System.out.println("Hello");
}).start();
```

On the Java Tutorials page for Lambda Expressions, read **Syntax of Lambda Expressions** (<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#syntax>).

There's no magic here: Java still doesn't have first-class functions. So you can only use a lambda when the Java compiler can verify two things:

1. It must be able to determine the type of the functional object the lambda will create. In this example, the compiler sees that the `Thread` constructor takes a `Runnable`, so it will infer that the type must be `Runnable`.
2. This inferred type must be *functional interface*: an interface with only one (abstract) method. In this example, `Runnable` indeed only has a single method — `void run()` — so the compiler knows the code in the body of the lambda belongs in the body of a `run` method of a new `Runnable` object.

Java provides some standard functional interfaces ([//docs.oracle.com/javase/8/docs/api/?java/util/function/package-summary.html](https://docs.oracle.com/javase/8/docs/api/?java/util/function/package-summary.html)) we can use to write code in the map/filter/reduce pattern, e.g.:

- `Function<T,R>` ([//docs.oracle.com/javase/8/docs/api/?java/util/function/Function.html](https://docs.oracle.com/javase/8/docs/api/?java/util/function/Function.html))
represents unary functions from `T` to `R`
- `BiFunction<T,U,R>` ([//docs.oracle.com/javase/8/docs/api/?java/util/function/BiFunction.html](https://docs.oracle.com/javase/8/docs/api/?java/util/function/BiFunction.html))
represents binary functions from `T × U` to `R`
- `Predicate<T>` ([//docs.oracle.com/javase/8/docs/api/?java/util/function/Predicate.html](https://docs.oracle.com/javase/8/docs/api/?java/util/function/Predicate.html))
represents functions from `T` to boolean

So we could implement map in Java like so:

```
/*
 * Apply a function to every element of a list.
 * @param f function to apply
 * @param list list to iterate over
 * @return [f(list[0]), f(list[1]), ..., f(list[n-1])]
 */
public static <T,R> List<R> map(Function<T,R> f, List<T> list) {
    List<R> result = new ArrayList<>();
    for (T t : list) {
        result.add(f.apply(t));
    }
    return result;
}
```

And here's an example of using map; first we'll write it using the familiar syntax:

```
// anonymous classes like this one are effectively lambda expressions
Function<String,String> toLowerCase = new Function<>() {
    public String apply(String s) { return s.toLowerCase(); }
};
map(toLowerCase, Arrays.asList(new String[] {"A", "b", "C"}));
```

And with a lambda expression:

```
map(s -> s.toLowerCase(), Arrays.asList(new String[] {"A", "b", "C"}));
// --or--
map((s) -> s.toLowerCase(), Arrays.asList(new String[] {"A", "b", "C"}));
// --or--
map((s) -> { return s.toLowerCase(); }, Arrays.asList(new String[] {"A", "b", "C"}));
```

In this example, the lambda expression is just wrapping a call to `String`'s `toLowerCase`. We can use a *method reference* to avoid writing the lambda, with the syntax `::`. The signature of the method we refer to must match the signature required by the functional interface for static typing to be satisfied:

```
map(String::toLowerCase, Arrays.asList(new String[] {"A", "b", "C"}));
```

In the Java Tutorials, you can read more about **method references**

(<https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html>) if you want the details.

Using a method reference (vs. calling it) in Java serves the same purpose as referring to a function by name (vs. calling it) in Python.

Map/filter/reduce in Java

The abstract sequence type we defined above exists in Java as `Stream`

([//docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html](https://docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html)) , which defines `map` , `filter` , `reduce` , and many other operations.

Collection types like `List` and `Set` provide a `stream()` operation that returns a `Stream` for the collection, and there's an `Arrays.stream` function for creating a `Stream` from an array.

Here's one implementation of `allFilesIn` in Java with map and filter:

```
public class Words {
    static Stream<File> allFilesIn(File folder) {
        File[] children = folder.listFiles();
        Stream<File> descendants = Arrays.stream(children)
            .filter(File::isDirectory)
            .flatMap(Words::allFilesIn);
        return Stream.concat(descendants,
            Arrays.stream(children).filter(File::isFile));
    }
}
```

The map-and-flatten pattern is so common that Java provides a `flatMap` operation to do just that, and we've used it instead of defining `flatten` .

Here's `endsWith` :

```
static Predicate<File> endsWith(String suffix) {
    return f -> f.getPath().endsWith(suffix);
}
```

Given a `Stream<File>` `files` , we can now write e.g. `files.filter(endsWith(".java"))` to obtain a new filtered stream.

Look at the **revised Java code for this example** (<https://github.com/mit6005/sp16-ex25-words/blob/master/src/words/Words3.java>) .

You can compare **all three versions** (<https://github.com/mit6005/sp16-ex25-words/tree/master/src/words>) : the familiar Java implementation, Python with map/filter/reduce, and Java with map/filter/reduce.

Higher-order functions in Java

`Map/filter/reduce` are of course higher-order functions; so is `endsWith` above. Let's look at two more that we saw before: `compose` and `chain` .

The `Function` interface provides `compose` — but the implementation is very straightforward. In particular, once you get the types of the arguments and return values correct, Java's static typing makes it pretty much impossible to get the method body wrong:

```
/**
 * Compose two functions.
 * @param f function A->B
 * @param g function B->C
 * @return new function A->C formed by composing f with g
 */
public static <A,B,C> Function<A,C> compose(Function<A,B> f,
                                              Function<B,C> g) {
    return t -> g.apply(f.apply(t));
    // --or--
    // return new Function<A,C>() {
    //     public C apply(A t) { return g.apply(f.apply(t)); }
    // };
}
```

It turns out that we *can't* write `chain` in strongly-typed Java, because `List`'s (and other collections) must be homogeneous — we can specify a list whose elements are all of type `Function<A,B>`, but not one whose first element is a `Function<A,B>`, second is a `Function<B,C>`, and so on.

But here's `chain` for functions of the same input/output type:

```
/**
 * Compose a chain of functions.
 * @param funcs list of functions A->A to compose
 * @return function A->A made by composing list[0] ... list[n-1]
 */
public static <A> Function<A,A> chain(List<Function<A,A>> funcs) {
    return funcs.stream().reduce(Function.identity(), Function::compose);
}
```

Our Python version didn't use an initial value in the `reduce`, it required a non-empty list of functions. In Java, we've provided the identity function (that is, $f(t) = t$) as the identity value for the reduction.

Summary

This reading is about modeling problems and implementing systems with *immutable data* and operations that implement *pure functions*, as opposed to *mutable data* and operations with *side effects*. *Functional programming* is the name for this style of programming.

Functional programming is much easier to do when you have *first-class functions* in your language and you can build *higher-order functions* that abstract away control flow code.

Some languages — Haskell ([//www.haskell.org/](http://www.haskell.org/)), Scala ([//www.scala-lang.org/](http://www.scala-lang.org/)), OCaml ([//ocaml.org/](http://ocaml.org/)) — are strongly associated with functional programming. Many other languages — JavaScript (<https://developer.mozilla.org/en-US/docs/JavaScript>), Swift (<https://developer.apple.com/swift/>), several ([//msdn.microsoft.com/en-us/library/67ef8sbd.aspx](http://msdn.microsoft.com/en-us/library/67ef8sbd.aspx)).NET ([//fsharp.org](http://fsharp.org)) languages ([//msdn.microsoft.com/en-us/vstudio/hh388573.aspx](http://msdn.microsoft.com/en-us/vstudio/hh388573.aspx)), Ruby ([//www.ruby-lang.org/](http://www.ruby-lang.org/)), and so on — use functional programming to a greater or lesser extent. With Java's recently-added functional language features, if you continue programming in Java you should expect to see more functional programming there, too.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 26: Little Languages

Representing code as data

Building languages to solve problems

Music language

To be continued

Reading 26: Little Languages

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

In this reading we will begin to explore the design of a **little language** for constructing and manipulating music. Here's the bottom line: when you need to solve a problem, instead of writing a *program* to solve just that one problem, build a *language* that can solve a range of related problems.

The goal for this reading is to introduce the idea of **representing code as data** and familiarize you with an initial version of the **music language**.

Representing code as data

Recall the `Formula` datatype from *Recursive Data Types* (./16-recursive-data-types/recursive/#another_example_boolean_formulas) :

```
Formula = Variable(name:String)
      + Not(formula:Formula)
      + And(left:Formula, right:Formula)
      + Or(left:Formula, right:Formula)
```

We used instances of `Formula` to take propositional logic formulas, e.g. $(p \vee q) \wedge (\neg p \vee r)$, and represent them in a data structure, e.g.:

```
And(Or(Variable("p"), Variable("q")),
    Or(Not(Variable("p")), Variable("r")))
```

In the parlance of grammars and parsers, formulas are a *language*, and `Formula` is an *abstract syntax tree* (./18-parser-generators).

But why did we define a `Formula` type? Java already has a way to represent expressions of Boolean variables with *logical and*, *or*, and *not*. For example, given boolean variables `p`, `q`, and `r`:

```
(p || q) && ((!p) || r)
```

Done!

The answer is that the Java code expression `(p || q) && ((!p) || r)` is evaluated as soon as we encounter it in our running program. The `Formula` value `And(Or(...), Or(...))` is a **first-class value** that can be stored, passed and returned from one method to another, manipulated, and evaluated now or later (or more than once) as needed.

The `Formula` type is an example of **representing code as data**, and we've seen many more.

Consider this functional object ([./25-map-filter-reduce/#first-class_functions_in_java](#)) :

```
class VariableNameComparator implements Comparator<Variable> {
    public int compare(Variable v1, Variable v2) {
        return v1.name().compareTo(v2.name());
    }
}
```

An instance of `VariableNameComparator` is a value that can be passed around, returned, and stored. But at any time, the function that it represents can be invoked by calling its `compare` method with a couple of `Variable` arguments:

```
Variable v1, v2;
Comparator<Variable> c = new VariableNameComparator();
...
int a = c.compare(v1, v2);
int b = c.compare(v2, v1);
SortedSet<Variable> vars = new TreeSet<>(c); // vars is sorted by name
```

Lambda expressions allow us to create functional objects with a compact syntax:

```
Comparator<Variable> c = (v1, v2) -> v1.name().compareTo(v2.name());
```

Building languages to solve problems

When we define an abstract data type ([./12-abstract-data-types/](#)), we're extending the universe of built-in types provided by Java to include a new type, with new operations, appropriate to our problem domain. This new type is like a new language: a new set of nouns (values) and verbs (operations) we can manipulate. Of course, those nouns and verbs are abstractions built on top the existing nouns and verbs which were themselves already abstractions.

A *language* has greater flexibility than a mere *program*, because we can use a language to solve a large class of related problems, instead of just a single problem.

- That's the difference between writing `(p || q) && ((!p) || r)` and devising a `Formula` type to represent the semantically-equivalent Boolean formula.
- And it's the difference between writing a matrix multiplication function and devising a `MatrixExpression` type ([./16-recursive-data-types/matexpr/](#)) to represent matrix multiplications — and store them, manipulate them, optimize them, evaluate them, and so on.

First-class functions and functional objects enable us to create particularly powerful languages because we can capture patterns of computation as reusable abstractions.

Music language

In class, we will design and implement a language for generating and playing music. To prepare, let's first understand the Java APIs for playing music with the MIDI (<https://en.wikipedia.org/wiki/MIDI>) synthesizer. We'll see how to write a *program* to play MIDI music. Then we'll begin to develop our music *language* by writing a recursive abstract data type for simple musical tunes. We'll choose a notation for writing music in strings, and we'll implement a parser to create instances of our `Music` type.

The full source code for the basic music language (<https://github.com/mit6005/sp16-ex26-music-starting>) is on GitHub.

Clone the `sp16-ex26-music-starting` (<https://github.com/mit6005/sp16-ex26-music-starting>) repo so you can run the code and follow the discussion below.

Playing MIDI music

`music.midi.MidiSequencePlayer` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/midi/MidiSequencePlayer.java>) uses the Java MIDI APIs to play sequences of notes. It's quite a bit of code, and you don't need to understand how it works.

`MidiSequencePlayer` implements the `music.SequencePlayer` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/SequencePlayer.java>) interface, allowing clients to use it without depending on the particular MIDI implementation. We *do* need to understand this interface and the types it depends on:

`addNote : SequencePlayer × Instrument × Pitch × double × double → void` ([SequencePlayer.java:15](https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/SequencePlayer.java#L15) (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/SequencePlayer.java#L15>)) is the workhorse of our music player. Calling this method schedules a musical pitch to be played at some time during the piece of music.

`play : SequencePlayer → void` ([SequencePlayer.java:20](https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/SequencePlayer.java#L20) (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/SequencePlayer.java#L20>)) actually plays the music. Until we call this method, we're just scheduling music that will, eventually, be played.

The `addNote` operation depends on two more types:

`Instrument` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Instrument.java>) is an enumeration of all the available MIDI instruments.

`Pitch` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Pitch.java>) is an abstract data type for musical pitches (think keys on the piano keyboard).

Read and understand the `Pitch` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Pitch.java>) documentation and the specifications for its `public constructor` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Pitch.java#L57-68>) and all its `public methods` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Pitch.java#L70-122>).

Our music data type will rely on `Pitch` in its rep, so be sure to understand the `Pitch` spec as well as its rep and abstraction function.

Using the MIDI sequence player and `Pitch`, we're ready to write code for our first bit of music!

Read and understand the `music.examples.ScaleSequence` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/examples/ScaleSequence.java>) code.

Run the main method in `ScaleSequence`. You should hear a one-octave scale!

Music data type

The `Pitch` datatype is useful, but if we want to represent a whole piece of music using `Pitch` objects, we should create an abstract data type to encapsulate that representation.

To start, we'll define the `Music` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Music.java>) type with a few operations:

`notes : String × Instrument → Music` (`MusicLanguage.java:51` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L51>)) makes a new `Music` from a string of simplified abc notation, described below.

`duration : Music → double` (`Music.java:11` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Music.java#L11>)) returns the duration, in beats, of the piece of music.

`play : Music × SequencePlayer × double → void` (`Music.java:18` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Music.java#L18>)) plays the piece of music using the given sequence player.

We'll implement `duration` and `play` as instance methods of `Music`, so we declare them in the `Music` interface.

`notes` will be a static factory method; rather than put it in `Music` (which we could do), we'll put it in a separate class: `MusicLanguage` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java>) will be our place for all the static methods we write to operate on `Music`.

Now that we've chosen some operations in the spec of `Music`, let's choose a representation.

- Looking at `ScaleSequence` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/examples/ScaleSequence.java>), the first concrete variant that might jump out at us is one to capture the information in each call to `addNote`: a particular pitch on a particular instrument played for some amount of time. We'll call this a `Note` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Note.java>).
- The other basic element of music is the silence between notes: `Rest` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Rest.java>).
- Finally, we need a way to glue these basic elements together into larger pieces of music. We'll choose a tree-like structure: `Concat(m1, m2:Music)` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Concat.java>) represents `m1` followed by `m2`, where `m1` and `m2` are any music.

This tree structure turns out to be an elegant decision as we further develop our `Music` type later on. In a real design process, we might iterate on the recursive structure of `Music` before we find the best implementation.

Here's the datatype definition:

```
Music = Note(duration:double, pitch:Pitch, instrument:Instrument)
      + Rest(duration:double)
      + Concat(m1:Music, m2:Music)
```

Composite

`Music` is an example of the **composite pattern**, in which we treat both single objects (*primitives*, e.g. `Note` and `Rest`) and groups of objects (*composites*, e.g. `Concat`) the same way.

- `Formula` is also an example of the composite pattern.
- The GUI view tree relies heavily on the composite pattern: there are *primitive views* like `JLabel` and `JTextField` that don't have children, and *composite views* like `JPanel` and `JScrollPane` that do contain other views as children. Both implement the common `JComponent` interface.

The composite pattern gives rise to a tree data structure, with primitives at the leaves and composites at the internal nodes.

Emptiness

One last design consideration: how do we represent the empty music? It's always good to have a representation for *nothing*, and we're certainly not going to use `null`.

We could introduce an `Empty` variant, but instead we'll use a `Rest` of duration `0` to represent emptiness.

Implementing basic operations

First we need to create the `Note` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Note.java>), `Rest` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Rest.java>), and `Concat` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Concat.java>) variants. All three are straightforward to implement, starting with constructors, `checkRep`, some observers, `toString`, and the equality methods.

- Since the `duration` operation is an instance method, each variant implements `duration` appropriately.
- The `play` operation is also an instance method; we'll discuss it below under *implementing the player*.

And we'll discuss the `notes` operation in *implementing the parser*.

Read and understand the `Note` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Note.java>), `Rest` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Rest.java>), and `Concat` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Concat.java>) classes.

To avoid representation exposure, let's add some additional static factory methods to the `Music` interface:

```
note : double × Pitch × Instrument → Music ( MusicLanguage.java:92
(https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L92)
)
```

```
rest : double → Music ( MusicLanguage.java:100 (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L100) )
```

`concat : Music × Music → Music` (`MusicLanguage.java:113` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L113>)) is our first producer operation.

All three of them are easy to implement by constructing the appropriate variant.

Music notation

We will write pieces of music using a simplified version of **abc notation** ([//en.wikipedia.org/wiki/ABC_notation](https://en.wikipedia.org/wiki/ABC_notation)) , a text-based music format.

We've already been representing pitches using their familiar letters. Our simplified abc notation represents sequences of **notes** and **rests** with syntax for indicating their **duration** , **accidental** (sharp or flat), and **octave** .

For example:

C D E F G A B C' B A G F E D C represents the one-octave ascending and descending C major scale we played in `ScaleSequence` . C is middle C, and C' is C one octave above middle C. Each note is a quarter note.

C/2 D/2 _E/2 F/2 G/2 _A/2 _B/2 C' is the ascending scale in C minor, played twice as fast. The E, A, and B are flat. Each note is an eighth note.

Read and understand the specification of **notes** in `MusicLanguage` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L20-50>) .

You don't need to understand the parser implementation yet, but you should understand the simplified abc notation enough to make sense of the examples.

If you're not familiar with music theory — why is an octave 8 notes but only 12 semitones? — don't worry. You might not be able to look at the abc strings and guess what they sound like, but you can understand the point of choosing a convenient textual syntax.

Implementing the parser

The `notes` method parses strings of simplified abc notation into `Music` .

notes : String × Instrument → Music (`MusicLanguage.java:51` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L51>)) splits the input into individual symbols (e.g. A,,/2 , .1/2). We start with the empty `Music` , `rest(0)` , symbols are parsed individually, and we build up the `Music` using `concat` .

parseSymbol : String × Instrument → Music (`MusicLanguage.java:62` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L62>)) returns a `Rest` or a `Note` for a single abc symbol (`symbol` in the grammar). It only parses the type (rest or note) and duration; it relies on `parsePitch` to handle pitch letters, accidentals, and octaves.

parsePitch : String → Pitch (`MusicLanguage.java:77` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L77>)) returns a `Pitch` by parsing a pitch grammar production. You should be able to understand the recursion — what's the base case? What are the recursive cases?

Implementing the player

Recall our operation for playing music:

play : Music × SequencePlayer × double → void (`Music.java:18` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Music.java#L18>)) plays the piece of music using the given sequence player after the given number of beats delay.

Why does this operation take `atBeat` ? Why not simply play the music now ?

If we define `play` in that way, we won't be able to play sequences of notes over time unless we actually `pause` during the `play` operation, for example with `Thread.sleep`. Our sequence player's `addNote` operation (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/SequencePlayer.java#L15>) is already designed to schedule notes in the future — it handles the delay.

With that design decision, it's straightforward to implement `play` in every variant of `Music`.

Read and understand the `Note.play` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Note.java#L55>), `Rest.play` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Rest.java#L33>), and `Concat.play` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Concat.java#L51>) methods.

You should be able to follow their recursive implementations.

Just one more piece of utility code before we're ready to jam: `music.midi.MusicPlayer` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/midi/MusicPlayer.java>) plays a `Music` using the `MidiSequencePlayer`. `Music` doesn't know about the concrete type of the sequence player, so we need a bit of code to bring them together.

Bringing this *all* together, let's use the `Music` ADT:

Read and understand the `music.examples.ScaleMusic` (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/examples/ScaleMusic.java>) code.

Run the main method in `ScaleMusic`. You should hear the same one-octave scale again.

That's not very exciting, so **read `music.examples.RowYourBoatInitial`** (<https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/examples/RowYourBoatInitial.java>) and **run the main method**. You should hear *Row, row, row your boat!*

Can you follow the flow of the code from calling `notes(...)` to having an instance of `Music` to the recursive `play(...)` call to individual `addNote(...)` calls?

To be continued

Playing *Row, row, row your boat* is pretty exciting, but so far the most powerful thing we've done is not so much the *music language* as it is the very basic *music parser*. Writing music using the simplified abc notation is clearly much more **easy to understand**, **safe from bugs**, and **ready for change** than writing page after page of `addNote addNote addNote ...`

In class, we'll expand our music language and turn it into a powerful tool for constructing and manipulating complex musical structures.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 27: Team Version Control

Git workflow

Viewing commit history

Graph of commits

Using version control in a team

Team project

Reading 27: Team Version Control

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- Review Git basics and the commit graph
- Practice multi-user Git scenarios

Git workflow

You've been using Git for problem sets and in-class exercises for a while now. Most of the time, you haven't had to coordinate with other people pushing and pulling to and from the same repository as you at the same time. For the group projects, that will change.

In this reading, prepare for some in-class Git exercises by reviewing what you know and brushing up on some commands. Now that you're more comfortable with Git basics, it's a good time to go back and review some of the resources from the beginning of the semester.

Review **Inventing version control** ([..../05-version-control/#inventing_version_control](#)) : one developer, multiple developers, and branches.

If you need to, review **Learn the Git workflow** ([..../getting-started/#git](#)) from the *Getting Started* page.

Viewing commit history

Review **2.3 Viewing the Commit History** ([//git-scm.com/book/en/Git-Basics-Viewing-the-Commit-History](#)) from *Pro Git*.

You don't need to remember all the different command-line options presented in the book! Instead, learn what's possible so you know what to search for when you need it.

Clone the example repo from **Version Control** (`./05-version-control/#copy_an_object_graph_with_git_clone`) :
<https://github.com/mit6005/sp16-ex05-hello-git.git>

Use `log` commands to make sure you understand the history of the repo.

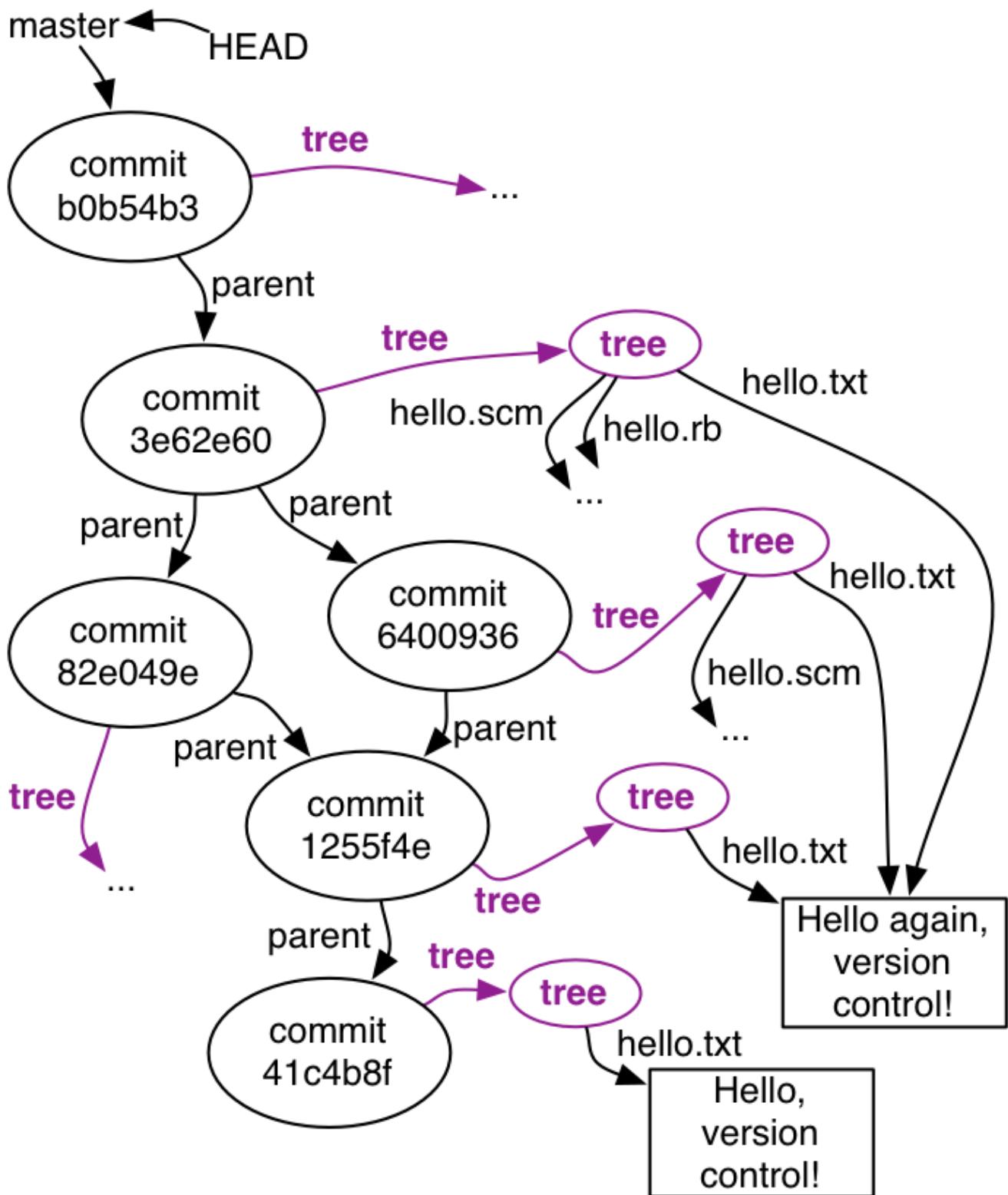
Graph of commits

Recall that the history recorded in a Git repository is a directed acyclic graph. The history of any particular branch in the repo (such as the default `master` branch) starts at some initial commit, and then its history may split apart and come back together, if multiple developers made changes in parallel (or if a single developer worked on two different machines without committing-pushing-pulling before the switch).

Here's the output of `git log` (`../../getting-started/#config-git`) for the example repository, which shows an ASCII-art graph:

```
* b0b54b3 (HEAD, origin/master, origin/HEAD, master) Greeting in Java
*   3e62e60 Merge
 |\ 
| * 6400936 Greeting in Scheme
* | 82e049e Greeting in Ruby
|/
* 1255f4e Change the greeting
* 41c4b8f Initial commit
```

And here is a diagram of the DAG:



In the `ex05-hello-git` example repo, make sure you can explain where the history of `master` splits apart, and where it comes back together.

Review **Merging** (`..../05-version-control/#merging`) from the *Version Control* reading.

You should understand every step of the process, and how it relates to the result in the example repo.

Review the **Getting Started section on merges** (`..../getting-started/#merges`), including merging and merge conflicts.

Using version control in a team

Every team develops its own standards for version control, and the size of the team and the project they're working on is a major factor. Here are some guidelines for a small-scope team project of the kind you will undertake in 6.005:

- **Communicate.** Tell your teammates what you're going to work on. Tell them that you're working on it. And tell them that you worked on it. Communication is the best way to avoid wasted time and effort cleaning up broken code.
- **Write specs.** Necessary for the things we care about in 6.005, and part of good communication.
- **Write tests.** Don't wait for a giant pile of code to accumulate before you try to test it. Avoid having one person write tests while another person writes implementation (unless the implementation is a prototype you plan to throw away). Write tests first to make sure you agree on the specs. Everyone should take responsibility for the correctness of their code.
- **Run the tests.** Tests can't help you if you don't run them. Run them before you start working, run them again before you commit.
- **Automate.** You've already automated your tests with a tool like JUnit, but now you want to automate running those tests whenever the project changes. For 6.005 group projects, we provide Didit as a way to automatically run your tests every time a team member pushes to Athena. This also removes "it worked on my machine" from the equation: either it works in the automated build, or it needs to be fixed.
- **Review what you commit.** Use `git diff --staged` or a GUI program to see what you're about to commit. Run the tests. Don't use `commit -a`, that's a great way to fill your repo with `println`s and other stuff you didn't mean to commit. Don't annoy your teammates by committing code that doesn't compile, spews debug output, isn't actually used, etc.
- **Pull before you start working.** Otherwise, you probably don't have the latest version as your starting point — you're editing an old version of the code! You're guaranteed to have to merge your changes later, and you're in danger of having to waste time resolving a merge conflict.
- **Sync up.** At the end of a day or at the end of a work session, make sure everyone has pushed and pulled all the changes, you're all at the same commit, and everyone is satisfied with the state of the project.

We don't recommend using features like branching or rebasing for 6.005-sized projects.

We do strongly recommend working together in the same place at the same time, especially if this is your first group software engineering experience.

Team project

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

Reading 2: Basic Java

Objectives

- Learn basic Java syntax and semantics
- Transition from writing Python to writing Java

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Getting started with the Java Tutorials

The next few sections link to the [Java Tutorials](https://docs.oracle.com/javase/tutorial/index.html) ([//docs.oracle.com/javase/tutorial/index.html](https://docs.oracle.com/javase/tutorial/index.html)) to get you up to speed with the basics.

You can also look at Getting Started: Learning Java ([..../getting-started/java.html](https://docs.oracle.com/javase/tutorial/getStarted/intro/index.html)) as an alternative resource.

This reading and other resources will frequently refer you to the Java API documentation ([//docs.oracle.com/javase/8/docs/api/](https://docs.oracle.com/javase/8/docs/api/)) which describes all the classes built in to Java.

Language basics

Read [Language Basics](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html) ([//docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html)) .

You should be able to answer the questions on the *Questions and Exercises* pages for all four of the language basics topics.

- Questions: Variables
([//docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_variables.html](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_variables.html))
- Questions: Operators
([//docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_operators.html](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_operators.html))
- Questions: Expressions, Statements, Blocks
([//docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_expressions.html](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_expressions.html))

- Questions: Control Flow
https://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_flow.html

Note that each *Questions and Exercises* page has a link at the bottom to solutions.

Also check your understanding by answering some questions about how the basics of Java compare to the basics of Python:

Numbers and strings

Read **Numbers and Strings** (<https://docs.oracle.com/javase/tutorial/java/data/index.html>) .

Don't worry if you find the `Number` wrapper classes confusing. They are.

You should be able to answer the questions on both *Questions and Exercises* pages.

- Questions: Numbers (<https://docs.oracle.com/javase/tutorial/java/data/QandE/numbers-questions.html>)
- Questions: Characters, Strings (<https://docs.oracle.com/javase/tutorial/java/data/QandE/characters-questions.html>)

Classes and objects

Read **Classes and Objects** (<https://docs.oracle.com/javase/tutorial/java/javaOO/index.html>) .

You should be able to answer the questions on the first two *Questions and Exercises* pages.

- Questions: Classes (<https://docs.oracle.com/javase/tutorial/java/javaOO/QandE/creating-questions.html>)
- Questions: Objects (<https://docs.oracle.com/javase/tutorial/java/javaOO/QandE/objects-questions.html>)

Don't worry if you don't understand everything in *Nested Classes* and *Enum Types* right now. You can go back to those constructs later in the semester when we see them in class.

Hello, world!

Read **Hello World!** (<https://docs.oracle.com/javase/tutorial/getStarted/application/index.html>)

You should be able to create a new `HelloWorldApp.java` file, enter the code from that tutorial page, and compile and run the program to see `Hello World!` on the console.

Snapshot diagrams

It will be useful for us to draw pictures of what's happening at runtime, in order to understand subtle questions. **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 class and in 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 (with each other and with your TA).

- To pave the way for richer design notations in subsequent courses. For example, snapshot diagrams generalize into object models in 6.170.

Although the diagrams in this course use examples from Java, the notation can be applied to any modern programming language, e.g., Python, Javascript, C++, Ruby.

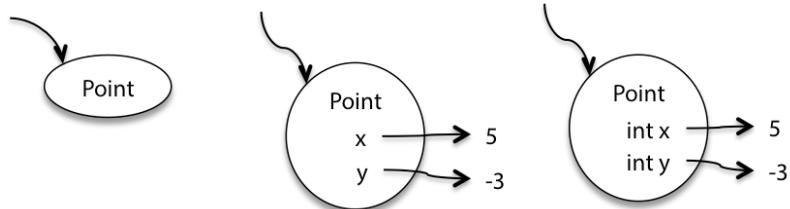
Primitive values

Primitive values are represented by bare constants. The incoming arrow is a reference to the value 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`, but both are fine.

Mutating values vs. reassigning variables

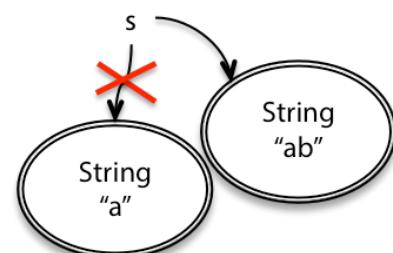
Snapshot diagrams give us a way to visualize the distinction between changing a variable and changing a value:

- When you assign to a variable or a field, 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.

Reassignment and immutable values

For example, if we have a `String` ([//docs.oracle.com/javase/8/docs/api/?java/lang/String.html](https://docs.oracle.com/javase/8/docs/api/?java/lang/String.html)) variable `s`, we can reassign it from a value of "a" to "ab".

```
String s = "a";
s = s + "b";
```



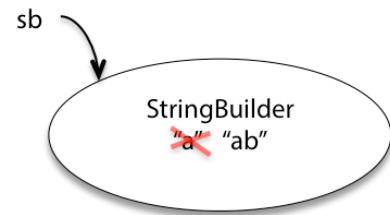
`String` is an example of an *immutable* type, a type whose values can never change once they have been created. Immutability (immunity from change) is a major design principle in this course, and we'll talk much more about it in future readings.

Immutable objects (intended by their designer to always represent the same value) are denoted in a snapshot diagram by a double border, like the `String` objects in our diagram.

Mutable values

By contrast, `StringBuilder` ([//docs.oracle.com/javase/8/docs/api/?java/lang/StringBuilder.html](https://docs.oracle.com/javase/8/docs/api/?java/lang/StringBuilder.html)) (another built-in Java class) is a *mutable* object that represents a string of characters, and it has methods that change the value of the object:

```
StringBuilder sb = new StringBuilder("a");
sb.append("b");
```



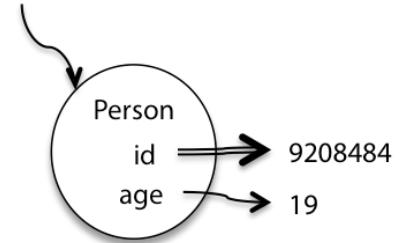
These two snapshot diagrams look very different, which is good: the difference between mutability and immutability will play an important role in making our code safe from bugs .

Immutable references

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.



In a snapshot diagram, an immutable reference (`final`) is denoted by a double arrow. Here's an object whose `id` never changes (it can't be reassigned to a different number), but whose `age` can change.

Notice that we can have an *immutable reference* to a *mutable value* (for example: `final StringBuilder sb`) whose value can change even though we're pointing to the same object.

We can also have a *mutable reference* to an *immutable value* (like `String s`), where the value of the variable can change because it can be re-pointed to a different object.

Java Collections

The very first Language Basics tutorial discussed **arrays**

([//docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html)), which are *fixed-length* containers for a sequence of objects or primitive values. Java provides a number of more powerful and flexible tools for managing *collections* of objects: the **Java Collections Framework** .

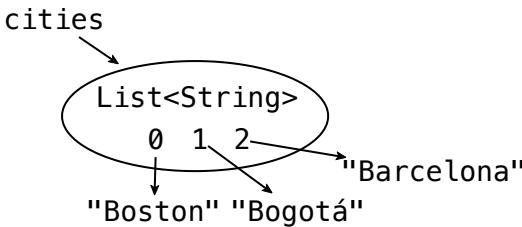
Lists, Sets, and Maps

A Java `List` ([//docs.oracle.com/javase/8/docs/api/?java/util/List.html](https://docs.oracle.com/javase/8/docs/api/?java/util/List.html)) is similar to a Python `list` (<https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range>) . A `List` contains an ordered collection of zero or more objects, where the same object might appear multiple times. We can add and remove items to and from the `List` , which will grow and shrink to accomodate its contents.

Example `List` operations:

Java	description	Python
<code>int count = lst.size();</code>	count the number of elements	<code>count = len(lst)</code>
<code>lst.add(e);</code>	append an element to the end	<code>lst.append(e)</code>
<code>if (lst.isEmpty()) ...</code>	test if the list is empty	<code>if not lst: ...</code>

In a snapshot diagram, we represent a `List` as an object with indices drawn as fields:



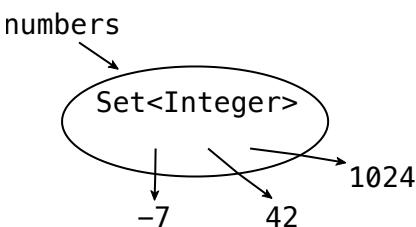
This list of `cities` might represent a trip from Boston to Bogotá to Barcelona.

A Set (<https://docs.oracle.com/javase/8/docs/api/?java/util/Set.html>) is an unordered collection of zero or more unique objects. Like a mathematical set or a Python set (<https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset>) – and unlike a `List` – an object cannot appear in a set multiple times. Either it's in or it's out.

Example Set operations:

Java	description	Python
<code>s1.contains(e)</code>	test if the set contains an element	<code>e in s1</code>
<code>s1.containsAll(s2)</code>	test whether $s1 \supseteq s2$	<code>s1.issuperset(s2)</code> <code>s1 >= s2</code>
<code>s1.removeAll(s2)</code>	remove $s2$ from $s1$	<code>s1.difference_update(s2)</code> <code>s1 -= s2</code>

In a snapshot diagram, we represent a `Set` as an object with no-name fields:



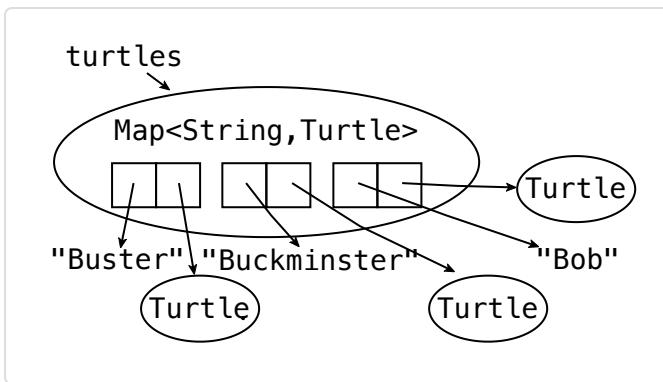
Here we have a set of integers, in no particular order: 42, 1024, and -7.

A Map (<https://docs.oracle.com/javase/8/docs/api/?java/util/Map.html>) is similar to a Python dictionary (<https://docs.python.org/3/library/stdtypes.html#mapping-types-dict>). In Python, the **keys** of a map must be hashable (<https://docs.python.org/3/glossary.html#term-hashable>). Java has a similar requirement that we'll discuss when we confront how equality works between Java objects.

Example Map operations:

Java	description	Python
<code>map.put(key, val)</code>	add the mapping $key \rightarrow val$	<code>map[key] = val</code>
<code>map.get(key)</code>	get the value for a key	<code>map[key]</code>
<code>map.containsKey(key)</code>	test whether the map has a key	<code>key in map</code>
<code>map.remove(key)</code>	delete a mapping	<code>del map[key]</code>

In a snapshot diagram, we represent a `Map` as an object that contains key/value pairs:



This `turtles` map contains `Turtle` objects assigned to `String` keys: `Bob`, `Buckminster`, and `Buster`.

Literals

Python provides convenient syntax for creating lists:

```
lst = [ "a", "b", "c" ]
```

And maps:

```
map = { "apple": 5, "banana": 7 }
```

Java does not. It does provide a literal syntax for arrays:

```
String[] arr = { "a", "b", "c" };
```

But this creates an `array`, not a `List`. We can use a provided utility function ([//docs.oracle.com/javase/8/docs/api/?java/util/Arrays.html](https://docs.oracle.com/javase/8/docs/api/?java/util/Arrays.html)) to create a `List` from the array:

```
Arrays.asList(new String[] { "a", "b", "c" })
```

A `List` created with `Arrays.asList` does come with a restriction: its length is fixed.

Generics: declaring List, Set, and Map variables

Unlike Python collection types, with Java collections we can restrict the type of objects contained in the collection. When we add an item, the compiler can perform *static checking* to ensure we only add items of the appropriate type. Then, when we pull out an item, we are guaranteed that its type will be what we expect.

Here's the syntax for declaring some variables to hold collections:

```
List<String> cities;           // a List of Strings
Set<Integer> numbers;         // a Set of Integers
Map<String,Turtle> turtles;   // a Map with String keys and Turtle values
```

Because of the way generics work, we cannot create a collection of primitive types. For example, `Set<int>` does *not* work. However, as we saw earlier, `int`'s have an `Integer` wrapper we can use (e.g. `Set<Integer> numbers`).

In order to make it easier to use collections of these wrapper types, Java does some automatic conversion. If we have declared `List<Integer> sequence`, this code works:

```
sequence.add(5);           // add 5 to the sequence
int second = sequence.get(1); // get the second element
```

ArrayLists and LinkedLists: creating Lists

As we'll see soon enough, Java helps us distinguish between the *specification* of a type – what does it do? – and the *implementation* – what is the code?

`List`, `Set`, and `Map` are all *interfaces*: they define how these respective types work, but they don't provide implementation code. There are several advantages, but one potential advantage is that we, the users of these types, get to choose different implementations in different situations.

Here's how to create some actual `List`s:

```
List<String> firstNames = new ArrayList<String>();
List<String> lastNames = new LinkedList<String>();
```

If the generic type parameters are the same on the left and right, Java can infer what's going on and save us some typing:

```
List<String> firstNames = new ArrayList<>();
List<String> lastNames = new LinkedList<>();
```

`ArrayList` ([//docs.oracle.com/javase/8/docs/api/?java/util/ArrayList.html](https://docs.oracle.com/javase/8/docs/api/?java/util/ArrayList.html)) and `LinkedList` ([//docs.oracle.com/javase/8/docs/api/?java/util/LinkedList.html](https://docs.oracle.com/javase/8/docs/api/?java/util/LinkedList.html)) are two implementations of `List`. Both provide all the operations of `List`, and those operations must work as described in the documentation for `List`. In this example, `firstNames` and `lastNames` will behave the same way; if we swapped which one used `ArrayList` vs. `LinkedList`, our code will not break.

Unfortunately, this ability to choose is also a burden: we didn't care how Python lists worked, why should we care whether our Java lists are `ArrayLists` or `LinkedLists`? Since the only difference is performance, for 6.005 we *don't*.

When in doubt, use `ArrayList`.

HashSets and HashMaps: creating Sets and Maps

`HashSet` ([//docs.oracle.com/javase/8/docs/api/?java/util/HashSet.html](https://docs.oracle.com/javase/8/docs/api/?java/util/HashSet.html)) is our default choice for `Set`s:

```
Set<Integer> numbers = new HashSet<>();
```

Java also provides sorted sets ([//docs.oracle.com/javase/8/docs/api/?java/util/SortedSet.html](https://docs.oracle.com/javase/8/docs/api/?java/util/SortedSet.html)) with the `TreeSet` ([//docs.oracle.com/javase/8/docs/api/?java/util/TreeSet.html](https://docs.oracle.com/javase/8/docs/api/?java/util/TreeSet.html)) implementation.

And for a `Map` the default choice is `HashMap` ([//docs.oracle.com/javase/8/docs/api/?java/util/HashMap.html](https://docs.oracle.com/javase/8/docs/api/?java/util/HashMap.html)):

```
Map<String, Turtle> turtles = new HashMap<>();
```

Iteration

So maybe we have:

```
List<String> cities      = new ArrayList<>();
Set<Integer> numbers    = new HashSet<>();
Map<String,Turtle> turtles = new HashMap<>();
```

A very common task is iterating through our cities/numbers/turtles/etc.

In Python:

```
for city in cities:
    print city

for num in numbers:
    print num

for key in turtles:
    print "%s: %s" % (key, turtles[key])
```

Java provides a similar syntax for iterating over the items in `List`s and `Set`s.

Here's the Java:

```
for (String city : cities) {
    System.out.println(city);
}

for (int num : numbers) {
    System.out.println(num);
}
```

We can't iterate over `Map`s themselves this way, but we can iterate over the keys as we did in Python:

```
for (String key : turtles.keySet()) {
    System.out.println(key + ": " + turtles.get(key));
}
```

Under the hood this kind of `for` loop uses an `Iterator` ([//docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html](https://docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html)) , a design pattern we'll see later in the class.

Iterating with indices

If you want to, Java provides different `for` loops that we can use to iterate through a list using its indices:

```
for (int ii = 0; ii < cities.size(); ii++) {
    System.out.println(cities.get(ii));
}
```

Unless we actually need the index value `ii` , this code is verbose and has more places for bugs to hide. Avoid.

Java API documentation

The previous section has a number of links to documentation for classes that are part of the Java platform API ([//docs.oracle.com/javase/8/docs/api/](https://docs.oracle.com/javase/8/docs/api/)).

API stands for *application programming interface*. If you want to program an app that talks to Facebook, Facebook publishes an API (more than one, in fact, for different languages and frameworks) you can program against. The Java API is a large set of generally useful tools for programming pretty much anything.

- **java.lang.String** ([//docs.oracle.com/javase/8/docs/api/?java/lang/String.html](https://docs.oracle.com/javase/8/docs/api/?java/lang/String.html)) is the full name for `String`. We can create objects of type `String` just by using "double quotes".
- **java.lang.Integer** ([//docs.oracle.com/javase/8/docs/api/?java/lang/Integer.html](https://docs.oracle.com/javase/8/docs/api/?java/lang/Integer.html)) and the other primitive wrapper classes. Java automatically converts between primitive and wrapped (or "boxed") types in most situations.
- **java.util.List** ([//docs.oracle.com/javase/8/docs/api/?java/util/List.html](https://docs.oracle.com/javase/8/docs/api/?java/util/List.html)) is like a Python list, but in Python, lists are part of the language. In Java, `List`s are implemented in... Java!
- **java.util.Map** ([//docs.oracle.com/javase/8/docs/api/?java/util/Map.html](https://docs.oracle.com/javase/8/docs/api/?java/util/Map.html)) is like a Python dictionary.
- **java.io.File** ([//docs.oracle.com/javase/8/docs/api/?java/io/File.html](https://docs.oracle.com/javase/8/docs/api/?java/io/File.html)) represents a file on disk. Take a look at the methods provided by `File`: we can test whether the file is readable, delete the file, see when it was last modified...
- **java.io.FileReader** ([//docs.oracle.com/javase/8/docs/api/?java/io/FileReader.html](https://docs.oracle.com/javase/8/docs/api/?java/io/FileReader.html)) lets us read text files.
- **java.io.BufferedReader** ([//docs.oracle.com/javase/8/docs/api/?java/io/BufferedReader.html](https://docs.oracle.com/javase/8/docs/api/?java/io/BufferedReader.html)) lets us read in text efficiently, and it also provides a very useful feature: reading an entire line at a time.

Let's take a closer look at the documentation for `BufferedReader`

([//docs.oracle.com/javase/8/docs/api/?java/io/BufferedReader.html](https://docs.oracle.com/javase/8/docs/api/?java/io/BufferedReader.html)). There are many things here that relate to features of Java we haven't discussed! Keep your head and focus on the **things in bold** below.

At the top of the page is the *class hierarchy* for `BufferedReader` and a list of *implemented interfaces*. A `BufferedReader` object has all of the methods of all those types (plus its own methods) available to use.

Next we see *direct subclasses*, and for an interface, *implementing classes*. This can help us find, for example, that `HashMap` ([//docs.oracle.com/javase/8/docs/api/?java/util/HashMap.html](https://docs.oracle.com/javase/8/docs/api/?java/util/HashMap.html)) is an implementation of `Map` ([//docs.oracle.com/javase/8/docs/api/?java/util/Map.html](https://docs.oracle.com/javase/8/docs/api/?java/util/Map.html)).

Next up: a **description of the class**. Sometimes these descriptions are a little obtuse, but **this is the first place you should go** to understand a class.

If you want to make a new `BufferedReader` the **constructor summary** is the first place to look.

The screenshot shows the Java API documentation for the `BufferedReader` class. At the top, there's a navigation bar with links for Overview, Package, Class (which is highlighted), Use, Tree, Deprecated, Index, and Help. Below the navigation bar, there are links for PREV CLASS, NEXT CLASS, FRAMES, NO FRAMES, and ALL CLASSES. The summary section includes links for SUMMARY: NESTED | FIELD | CONSTR | METHOD and DETAIL: FIELD | CONSTR | METHOD. The main content area starts with the `compact1`, `compact2`, and `compact3` methods from the `java.io` package. Then it defines the `Class BufferedReader`, which extends `java.lang.Object` and implements `java.io.Reader` and `java.io.BufferedReader`. It lists `Closeable`, `AutoCloseable`, and `Readable` as implemented interfaces, and `LineNumberReader` as a direct known subclass. The detailed description explains that `BufferedReader` reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. It notes that the buffer size may be specified, or the default size may be used. The buffer size is large enough for most purposes. It also states that in general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a `BufferedReader` around any Reader whose `read()` operations may be costly, such as `FileReaders` and `InputStreamReaders`. For example:

```
BufferedReader in
= new BufferedReader(new FileReader("foo.in"));
```

This will buffer the input from the specified file. Without buffering, each invocation of `read()` or `readLine()` could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient.

Programs that use `DataInputStreams` for textual input can be localized by replacing each `DataInputStream` with an appropriate `BufferedReader`.

Constructors aren't the only way to get a new object in Java, but they are the most common.

Next: the method summary lists all the methods we can call on a `BufferedReader` object.

Below the summary are detailed descriptions of each method and constructor. **Click a constructor or method to see the detailed description.** This is the first place you should go to understand what a method does.

Method Summary		
All Methods	Instance Methods	Concrete Methods
Modifier and Type	Method and Description	
void	<code>close()</code> Closes the stream and releases any system resources associated with it.	
<code>Stream<String></code>	<code>lines()</code> Returns a Stream, the elements of which are lines read from this BufferedReader.	
void	<code>mark(int readAheadLimit)</code> Marks the present position in the stream.	
boolean	<code>markSupported()</code> Tells whether this stream supports the mark() operation, which it does.	
int	<code>read()</code> Reads a single character.	
int	<code>read(char[] cbuf, int off, int len)</code> Reads characters into a portion of an array.	
<code>String</code>	<code>readLine()</code> Reads a line of text.	
boolean	<code>ready()</code> Tells whether this stream is ready to be read.	
void	<code>reset()</code> Resets the stream to the most recent mark.	
long	<code>skip(long n)</code> Skips characters.	
Methods inherited from class <code>java.io.Reader</code>		
<code>read(), read()</code>		
Methods inherited from class <code>java.lang.Object</code>		
<code>clone(), equals(), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(), wait(), wait()</code>		

Each detailed description includes:

- The **method signature** : we see the return type, the method name, and the parameters. We also see *exceptions* . For now, those usually mean errors the method can run into.
- The full **description** .
- **Parameters** : descriptions of the method arguments.
- And a description of what the method **returns** .

Specifications

These detailed descriptions are **specifications** . They allow us to use tools like `String` , `Map` , or `BufferedReader` *without* having to read or understand the code that implements them.

Reading, writing, understanding, and analyzing specifications will be one of our first major undertakings in 6.005, starting in a few classes.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 3: Testing

Validation

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Documenting Your Testing Strategy

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

Reading 3: Testing

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives for Today's Class

After today's class, you should:

- understand the value of testing, and know the process of test-first programming;
- be able to design a test suite for a method by partitioning its input and output space and choosing good test cases;
- be able to judge a test suite by measuring its code coverage; and
- understand and know when to use blackbox vs. whitebox testing, unit tests vs. integration tests, and automated regression testing.

Validation

Testing is an example of a more general process called *validation*. The purpose of validation is to uncover problems in a program and thereby increase your confidence in the program's correctness. Validation includes:

- **Formal reasoning** about a program, usually called *verification*. Verification constructs a formal proof that a program is correct. Verification is tedious to do by hand, and automated tool support for verification is still an active area of research. Nevertheless, small, crucial pieces of a program may be formally verified, such as the scheduler in an operating system, or the bytecode interpreter

- in a virtual machine, or the filesystem in an operating system ([//www.csail.mit.edu/crash_tolerant_data_storage](http://www.csail.mit.edu/crash_tolerant_data_storage)) .
- **Code review.** Having somebody else carefully read your code, and reason informally about it, can be a good way to uncover bugs. It's much like having somebody else proofread an essay you have written. We'll talk more about code review in the next reading.
 - **Testing** . Running the program on carefully selected inputs and checking the results.

Even with the best validation, it's very hard to achieve perfect quality in software. Here are some typical *residual defect rates* (bugs left over after the software has shipped) per kloc (one thousand lines of source code):

- 1 - 10 defects/kloc: Typical industry software.
- 0.1 - 1 defects/kloc: High-quality validation. The Java libraries might achieve this level of correctness.
- 0.01 - 0.1 defects/kloc: The very best, safety-critical validation. NASA and companies like Praxis can achieve this level.

This can be discouraging for large systems. For example, if you have shipped a million lines of typical industry source code (1 defect/kloc), it means you missed 1000 bugs!

Why Software Testing is Hard

Here are some approaches that unfortunately don't work well in the world of software.

Exhaustive testing is infeasible. The space of possible test cases 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!

Haphazard testing (“just try it and see if it works”) is less likely to find bugs, unless the program is so buggy that an arbitrarily-chosen input is more likely to fail than to succeed. It also doesn’t increase our confidence in program correctness.

Random or statistical testing doesn’t work well for software. Other engineering disciplines can test small random samples (e.g. 1% of hard drives manufactured) and infer the defect rate for the whole production lot. Physical systems can use many tricks to speed up time, like opening a refrigerator 1000 times in 24 hours instead of 10 years. These tricks give known failure rates (e.g. mean lifetime of a hard drive), but they assume continuity or uniformity across the space of defects. This is true for physical artifacts.

But it’s not true for software. Software behavior varies discontinuously and discretely across the space of possible inputs. The system may seem to work fine across a broad range of inputs, and then abruptly fail at a single boundary point. The famous Pentium division bug ([//www.willamette.edu/~mjaneba/pentprob.html](http://www.willamette.edu/~mjaneba/pentprob.html)) affected approximately 1 in 9 billion divisions. Stack overflows, out of memory errors, and numeric overflow bugs tend to happen abruptly, and always in the same way, not with probabilistic variation. That’s different from physical systems, where there is often visible evidence that the system is approaching a failure point (cracks in a bridge) or failures are distributed probabilistically near the failure point (so that statistical testing will observe some failures even before the point is reached).

Instead, test cases must be chosen carefully and systematically, and that’s what we’ll look at next.

Putting on Your Testing Hat

Testing requires having the right attitude. When you’re coding, your goal is to make the program work, but as a tester, you want to **make it fail** .

That's a subtle but important difference. It is all too tempting to treat code you've just written as a precious thing, a fragile eggshell, and test it very lightly just to see it work.

Instead, you have to be brutal. A good tester wields a sledgehammer and beats the program everywhere it might be vulnerable, so that those vulnerabilities can be eliminated.

Test-first Programming

Test early and often. Don't leave testing until the end, when you have a big pile of unvalidated code. Leaving testing until the end only makes debugging longer and more painful, because bugs may be anywhere in your code. It's far more pleasant to test your code as you develop it.

In test-first-programming, you write tests before you even write any code. The development of a single function proceeds in this order:

1. Write a specification for the function.
2. Write tests that exercise the specification.
3. Write the actual code. Once your code passes the tests you wrote, you're done.

The **specification** describes the input and output behavior of the function. It gives the types of the parameters and any additional constraints on them (e.g. `sqrt`'s parameter must be nonnegative). It also gives the type of the return value and how the return value relates to the inputs. You've already seen and used specifications on your problem sets in this class. In code, the specification consists of the method signature and the comment above it that describes what it does. We'll have much more to say about specifications a few classes from now.

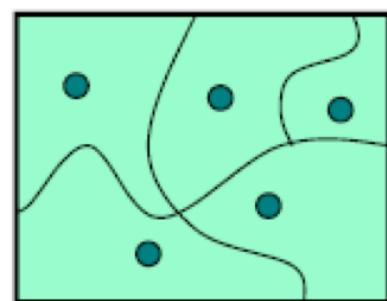
Writing tests first is a good way to understand the specification. The specification can be buggy, too — incorrect, incomplete, ambiguous, missing corner cases. Trying to write tests can uncover these problems early, before you've wasted time writing an implementation of a buggy spec.

Choosing Test Cases by Partitioning

Creating a good test suite is a challenging and interesting design problem. We want to pick a set of test cases that is small enough to run quickly, yet large enough to validate the program.

To do this, we divide the input space into **subdomains**, each consisting of a set of inputs. Taken together the subdomains completely cover the input space, so that every input lies in at least one subdomain. Then we choose one test case from each subdomain, and that's our test suite.

The idea behind subdomains is to partition the input space into sets of similar inputs on which the program has similar behavior. Then we use one representative of each set. This approach makes the best use of limited testing resources by choosing dissimilar test cases, and forcing the testing to explore parts of the input space that random testing might not reach.



We can also partition the output space into subdomains (similar outputs on which the program has similar behavior) if we need to ensure our tests will explore different parts of the output space. Most of the time, partitioning the input space is sufficient.

Example: `BigInteger.multiply()`

Let's look at an example. `BigInteger` (<https://docs.oracle.com/javase/8/docs/api/?java/math/BigInteger.html>) is a class built into the Java library that can represent integers of any size, unlike the primitive types `int` and `long` that have only limited ranges. `Biglnteger` has a method

`multiply` that multiplies two BigInteger values together:

```
/**  
 * @param val another BigInteger  
 * @return a BigInteger whose value is (this * val).  
 */  
public BigInteger multiply(BigInteger val)
```

For example, here's how it might be used:

```
BigInteger a = ...;  
BigInteger b = ...;  
BigInteger ab = a.multiply(b);
```

This example shows that even though only one parameter is explicitly shown in the method's declaration, `multiply` is actually a function of *two* arguments: the object you're calling the method on (`a` in the example above), and the parameter that you're passing in the parentheses (`b` in this example). In Python, the object receiving the method call would be explicitly named as a parameter called `self` in the method declaration. In Java, you don't mention the receiving object in the parameters, and it's called `this` instead of `self`.

So we should think of `multiply` as a function taking two inputs, each of type `BigInteger`, and producing one output of type `BigInteger`:

`multiply : BigInteger × BigInteger → BigInteger`

So we have a two-dimensional input space, consisting of all the pairs of integers (a,b). Now let's partition it. Thinking about how multiplication works, we might start with these partitions:

- a and b are both positive
- a and b are both negative
- a is positive, b is negative
- a is negative, b is positive

There are also some special cases for multiplication that we should check: 0, 1, and -1.

- a or b is 0, 1, or -1

Finally, as a suspicious tester trying to find bugs, we might suspect that the implementor of `BigInteger` might try to make it faster by using `int` or `long` internally when possible, and only fall back to an expensive general representation (like a list of digits) when the value is too big. So we should definitely also try integers that are very big, bigger than the biggest `long`.

- a or b is small
- the absolute value of a or b is bigger than `Long.MAX_VALUE`, the biggest possible primitive integer in Java, which is roughly 2^{63} .

Let's bring all these observations together into a straightforward partition of the whole (a,b) space. We'll choose a and b independently from:

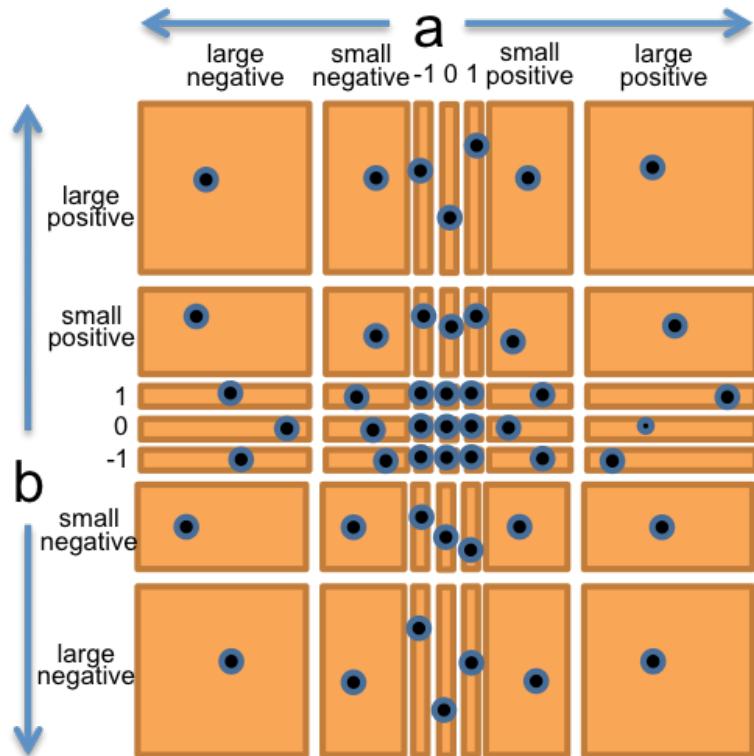
- 0
- 1
- -1
- small positive integer
- small negative integer
- huge positive integer
- huge negative integer

So this will produce $7 \times 7 = 49$ partitions that completely cover the space of pairs of integers.

To produce the test suite, we would pick an arbitrary pair (a,b) from each square of the grid, for example:

- $(a,b) = (-3, 25)$ to cover (small negative, small positive)
- $(a,b) = (0, 30)$ to cover (0, small positive)
- $(a,b) = (2^{100}, 1)$ to cover (large positive, 1)
- etc.

The figure at the right shows how the two-dimensional (a,b) space is divided by this partition, and the points are test cases that we might choose to completely cover the partition.



Example: `max()`

Let's look at another example from the Java library: the integer `max()` function, found in the `Math` class ([//docs.oracle.com/javase/8/docs/api/java/lang/Math.html](https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html)).

```
/**
 * @param a  an argument
 * @param b  another argument
 * @return the larger of a and b.
 */
public static int max(int a, int b)
```

Mathematically, this method is a function of the following type:

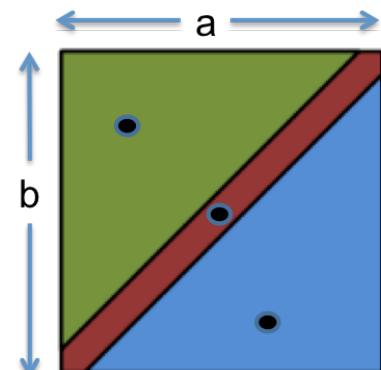
`max : int × int → int`

From the specification, it makes sense to partition this function as:

- $a < b$
- $a = b$
- $a > b$

Our test suite might then be:

- $(a, b) = (1, 2)$ to cover $a < b$
- $(a, b) = (9, 9)$ to cover $a = b$
- $(a, b) = (-5, -6)$ to cover $a > b$



Include Boundaries in the Partition

Bugs often occur at *boundaries* between subdomains. Some examples:

- 0 is a boundary between positive numbers and negative numbers

- the maximum and minimum values of numeric types, like `int` and `double`
- emptiness (the empty string, empty list, empty array) for collection types
- the first and last element of a collection

Why do bugs often happen at boundaries? One reason is that programmers often make **off-by-one mistakes** (like writing `<=` instead of `<`, or initializing a counter to 0 instead of 1). Another is that some boundaries may need to be handled as special cases in the code. Another is that boundaries may be places of discontinuity in the code's behavior. When an `int` variable grows beyond its maximum positive value, for example, it abruptly becomes a negative number.

It's important to include boundaries as subdomains in your partition, so that you're choosing an input from the boundary.

Let's redo `max : int × int → int`.

Partition into:

- *relationship between a and b*
 - $a < b$
 - $a = b$
 - $a > b$
- *value of a*
 - $a = 0$
 - $a < 0$
 - $a > 0$
 - $a = \text{minimum integer}$
 - $a = \text{maximum integer}$
- *value of b*
 - $b = 0$
 - $b < 0$
 - $b > 0$
 - $b = \text{minimum integer}$
 - $b = \text{maximum integer}$

Now let's pick test values that cover all these classes:

- (1, 2) covers $a < b$, $a > 0$, $b > 0$
- (-1, -3) covers $a > b$, $a < 0$, $b < 0$
- (0, 0) covers $a = b$, $a = 0$, $b = 0$
- (`Integer.MIN_VALUE`, `Integer.MAX_VALUE`) covers $a < b$, $a = \text{minint}$, $b = \text{maxint}$
- (`Integer.MAX_VALUE`, `Integer.MIN_VALUE`) covers $a > b$, $a = \text{maxint}$, $b = \text{minint}$

Two Extremes for Covering the Partition

After partitioning the input space, we can choose how exhaustive we want the test suite to be:

- **Full Cartesian product.**

Every legal combination of the partition dimensions is covered by one test case. This is what we did for the `multiply` example, and it gave us $7 \times 7 = 49$ test cases. For the `max` example that included boundaries, which has three dimensions with 3 parts, 5 parts, and 5 parts respectively, it would mean up to $3 \times 5 \times 5 = 75$ test cases. In practice not all of these combinations are possible, however. For example, there's no way to cover the combination $a < b$, $a=0$, $b=0$, because `a` can't be simultaneously less than zero and equal to zero.

- **Cover each part.**

Every part of each dimension is covered by at least one test case, but not necessarily every combination. With this approach, the test suite for `max` might be as small as 5 test cases if

carefully chosen. That's the approach we took above, which allowed us to choose 5 test cases.

Often we strike some compromise between these two extremes, based on human judgement and caution, and influenced by whitebox testing and code coverage tools, which we look at next.

Blackbox and Whitebox Testing

Recall from above that the *specification* is the description of the function's behavior — the types of parameters, type of return value, and constraints and relationships between them.

Blackbox testing means choosing test cases only from the specification, not the implementation of the function. That's what we've been doing in our examples so far. We partitioned and looked for boundaries in `multiply` and `max` without looking at the actual code for these functions.

Whitebox testing (also called glass box testing) means choosing test cases with knowledge of how the function is actually implemented. For example, if the implementation selects different algorithms depending on the input, then you should partition according to those domains. If the implementation keeps an internal cache that remembers the answers to previous inputs, then you should test repeated inputs.

When doing whitebox testing, you must take care that your test cases don't *require* specific implementation behavior that isn't specifically called for by the spec. For example, if the spec says "throws an exception if the input is poorly formatted," then your test shouldn't check *specifically* for a `NullPointerException` just because that's what the current implementation does. The specification in this case allows *any* exception to be thrown, so your test case should likewise be general to preserve the implementor's freedom. We'll have much more to say about this in the class on specs.

Documenting Your Testing Strategy

For the example function on the left, on the right is how we can document the testing strategy we worked on in the partitioning exercises above. The strategy also addresses some boundary values we didn't consider before.

```
/**  
 * Reverses the end of a string.  
 *  
 * For example:  
 *   reverseEnd("Hello, world", 5)  
 *   returns "Hellodlrow ,"  
 *  
 * With start == 0, reverses the entire text.  
 * With start == text.length(), reverses nothing.  
 *  
 * @param text  non-null String that will have  
 *             its end reversed  
 * @param start the index at which the  
 *             remainder of the input is  
 *             reversed, requires  
 *             0 <= start <= text.length()  
 * @return input text with the substring from  
 *         start to the end of the string  
 *         reversed  
 */  
static String reverseEnd(String text, int start)
```

Document the strategy at the top of the test class:

```
/*  
 * Testing strategy  
 *  
 * Partition the inputs as follows:  
 * text.length(): 0, 1, > 1  
 * start:          0, 1, 1 < start < text.length(),  
 *                 text.length() - 1, text.length()  
 * text.length()-start: 0, 1, even > 1, odd > 1  
 *  
 * Include even- and odd-length reversals because  
 * only odd has a middle element that doesn't move.  
 *  
 * Exhaustive Cartesian coverage of partitions.  
 */
```

Document how each test case was chosen, including white box tests:

```
// covers text.length() = 0,  
//           start = 0 = text.length(),  
//           text.length()-start = 0  
@Test public void testEmpty() {  
    assertEquals("", reverseEnd("", 0));  
}  
  
// ... other test cases ...
```

Coverage

One way to judge a test suite is to ask how thoroughly it exercises the program. This notion is called **coverage**. Here are three common kinds of coverage:

- **Statement coverage** : is every statement run by some test case?
- **Branch coverage** : for every `if` or `while` statement in the program, are both the true and the false direction taken by some test case?
- **Path coverage** : is every possible combination of branches — every path through the program — taken by some test case?

Branch coverage is stronger (requires more tests to achieve) than statement coverage, and path coverage is stronger than branch coverage. In industry, 100% statement coverage is a common goal, but even that is rarely achieved due to unreachable defensive code (like “should never get here” assertions). 100% branch coverage is highly desirable, and safety critical industry code has even more arduous criteria (e.g., “MCDC,” modified decision/condition coverage). Unfortunately 100% path coverage is infeasible, requiring exponential-size test suites to achieve.

A standard approach to testing is to add tests until the test suite achieves adequate statement coverage: i.e., so that every reachable statement in the program is executed by at least one test case. In practice, statement coverage is usually measured by a code coverage tool, which counts the number of times each statement is run by your test suite. With such a tool, white box testing is easy; you just measure the coverage of your black box tests, and add more test cases until all important statements are logged as executed.

A good code

Element	Coverage	Covered Instructions	Total Instructions
multipart	6.7 %	48	717
src	6.7 %	48	717
multipart	0.0 %	0	75
sequence	57.8 %	48	83
FileSequenceReader.java	37.5 %	21	56
FileSequenceReaderTest.java	100.0 %	27	27
ui	0.0 %	0	559

(figures/eclemma.png)

coverage tool for Eclipse is EclEmma ([//www.eclemma.org/](http://www.eclemma.org/)) , shown on the right.

Lines that have been executed by the test suite are colored green, and lines not yet covered are red. If you saw this result from your coverage tool, your next step would be to come up with a test case that causes the body of the while loop to execute, and add it to your test suite so that the red lines become green.

Unit Testing and Stubs

A well-tested program will have tests for every individual module (where a module is a method or a class) that it contains. A test that tests an individual module, in isolation if possible, is called a **unit test**. Testing modules in isolation leads to much easier debugging. When a unit test for a module fails, you can be more confident that the bug is found in that module, rather than anywhere in the program.

The opposite of a unit test is an **integration test**, which tests a combination of modules, or even the entire program. If all you have are integration tests, then when a test fails, you have to hunt for the bug. It might be anywhere in the program. Integration tests are still important, because a program can fail at the connections between modules. For example, one module may be expecting different inputs than it's actually getting from another module. But if you have a thorough set of unit tests that give you confidence in the correctness of individual modules, then you'll have much less searching to do to find the bug.

Suppose you're building a web search engine. Two of your modules might be `getWebPage()` , which downloads web pages, and `extractWords()` , which splits a page into its component words:

```
/** @return the contents of the web page downloaded from url
 */
public static String getWebPage(URL url) {...}

/** @return the words in string s, in the order they appear,
 *         where a word is a contiguous sequence of
 *         non-whitespace and non-punctuation characters
 */
public static List<String> extractWords(String s) { ... }
```

These methods might be used by another module `makeIndex()` as part of the web crawler that makes the search engine's index:

```
/** @return an index mapping a word to the set of URLs
 *         containing that word, for all webpages in the input set
 */
public static Map<String, Set<URL>> makeIndex(Set<URL> urls) {
    ...
    for (URL url : urls) {
        String page = getWebPage(url);
        List<String> words = extractWords(page);
        ...
    }
    ...
}
```

In our test suite, we would want:

- unit tests just for `getWebPage()` that test it on various URLs
- unit tests just for `extractWords()` that test it on various strings
- unit tests for `makeIndex()` that test it on various sets of URLs

One mistake that programmers sometimes make is writing test cases for `extractWords()` in such a way that the test cases depend on `getWebPage()` to be correct. It's better to think about and test `extractWords()` in isolation, and partition it. Using test partitions that involve web page content might be reasonable, because that's how `extractWords()` is actually used in the program. But don't actually call `getWebPage()` from the test case, because `getWebPage()` may be buggy! Instead, store web page content as a literal string, and pass it directly to `extractWords()`. That way you're writing an isolated unit test, and if it fails, you can be more confident that the bug is in the module it's actually testing, `extractWords()`.

Note that the unit tests for `makeIndex()` can't easily be isolated in this way. When a test case calls `makeIndex()`, it is testing the correctness of not only the code inside `makeIndex()`, but also all the methods called by `makeIndex()`. If the test fails, the bug might be in any of those methods. That's why we want separate tests for `getWebPage()` and `extractWords()`, to increase our confidence in those modules individually and localize the problem to the `makeIndex()` code that connects them together.

Isolating a higher-level module like `makeIndex()` is possible if we write **stub** versions of the modules that it calls. For example, a stub for `getWebPage()` wouldn't access the internet at all, but instead would return mock web page content no matter what URL was passed to it. A stub for a class is often called a **mock object** ([//en.wikipedia.org/wiki/Mock_object](https://en.wikipedia.org/wiki/Mock_object)). Stubs are an important technique when building large systems, but we will generally not use them in 6.005.

Automated Testing and Regression Testing

Nothing makes tests easier to run, and more likely to be run, than complete automation. **Automated testing** means running the tests and checking their results automatically. A test driver should not be an interactive program that prompts you for inputs and prints out results for you to manually check. Instead, a test driver should invoke the module itself on fixed test cases and automatically check that the results are correct. The result of the test driver should be either “all tests OK” or “these tests failed: ...” A good testing framework, like JUnit, helps you build automated test suites.

Note that automated testing frameworks like JUnit make it easy to run the tests, but you still have to come up with good test cases yourself. *Automatic test generation* is a hard problem, still a subject of active computer science research.

Once you have test automation, it’s very important to rerun your tests when you modify your code. This prevents your program from *regressing* — introducing other bugs when you fix new bugs or add new features. Running all your tests after every change is called **regression testing**.

Whenever you find and fix a bug, take the input that elicited the bug and add it to your automated test suite as a test case. This kind of test case is called a *regression test*. This helps to populate your 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! Saving regression tests also protects against reversions that reintroduce the bug. The bug may be an easy error to make, since it happened once already.

This idea also leads to *test-first debugging*. When a bug arises, immediately write a test case for it that elicits it, and immediately add it to your test suite. Once you find and fix the bug, all your test cases will be passing, and you’ll be done with debugging and have a regression test for that bug.

In practice, these two ideas, automated testing and regression testing, are almost always used in combination.

Regression testing is only practical if the tests can be run often, automatically. Conversely, if you already have automated testing in place for your project, then you might as well use it to prevent regressions. So **automated regression testing** is a best-practice of modern software engineering.

Summary

In this reading, we saw these ideas:

- Test-first programming. Write tests before you write code.
- Partitioning and boundaries for choosing test cases systematically.
- White box testing and statement coverage for filling out a test suite.
- Unit-testing each module, in isolation as much as possible.
- Automated regression testing to keep bugs from coming back.

The topics of today’s reading connect to our three key properties of good software as follows:

- **Safe from bugs.** Testing is about finding bugs in your code, and test-first programming is about finding them as early as possible, immediately after you introduced them.
- **Easy to understand.** Testing doesn’t help with this as much as code review does.
- **Ready for change.** Readiness for change was considered by writing tests that only depend on behavior in the spec. We also talked about automated regression testing, which helps keep bugs from coming back when changes are made to code.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 4: Code Review

Code Review

Smelly Example #1

Don't Repeat Yourself

Comments Where Needed

Fail Fast

Avoid Magic Numbers

One Purpose For Each Variable

Smelly Example #2

Use Good Names

Use Whitespace to Help the Reader

Smelly Example #3

Don't Use Global Variables

Methods Should Return Results, not Print Them

Summary

Reading 4: Code Review

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives for Today's Class

In today's class, we will practice:

- code review: reading and discussing code written by somebody else
- general principles of good coding: things you can look for in every code review, regardless of programming language or program purpose

Code Review

Code review is careful, systematic study of source code by people who are not the original author of the code. It's analogous to proofreading a term paper.

Code review really has two purposes:

- **Improving the code.** Finding bugs, anticipating possible bugs, checking the clarity of the code, and checking for consistency with the project's style standards.
- **Improving the programmer.** Code review is an important way that programmers learn and teach each other, about new language features, changes in the design of the project or its coding standards, and new techniques. In open source projects, particularly, much conversation happens in the context of code reviews.

Code review is widely practiced in open source projects like Apache and Mozilla ([//blog.humphd.org/vocamus-1569/?p=1569](http://blog.humphd.org/vocamus-1569/?p=1569)) . Code review is also widely practiced in industry. At Google, you can't push any code into the main repository until another engineer has signed off on it in a code review.

In 6.005, we'll do code review on problem sets, as described in the Code Reviewing document ([..../general/code-review.html](#)) on the course website.

Style Standards

Most companies and large projects have coding style standards (for example, Google Java Style ([//google.github.io/styleguide/javaguide.html](http://google.github.io/styleguide/javaguide.html))) . These can get pretty detailed, even to the point of specifying whitespace (how deep to indent) and where curly braces and parentheses should go. These kinds of questions often lead to holy wars ([//www.outpost9.com/reference/jargon/jargon_23.html#TAG897](http://www.outpost9.com/reference/jargon/jargon_23.html#TAG897)) since they end up being a matter of taste and style.

For Java, there's a general style guide

([//www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html](http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html)) (unfortunately not updated for the latest versions of Java). Some of its advice gets very specific:

- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.

In 6.005, we have no official style guide of this sort. We're not going to tell you where to put your curly braces. That's a personal decision that each programmer should make. It's important to be self-consistent, however, and it's very important to follow the conventions of the project you're working on. If you're the programmer who reformats every module you touch to match your personal style, your teammates will hate you, and rightly so. Be a team player.

But there are some rules that are quite sensible and target our big three properties, in a stronger way than placing curly braces. The rest of this reading talks about some of these rules, at least the ones that are relevant at this point in the course, where we're mostly talking about writing basic Java. These are some things you should start to look for when you're code reviewing other students, and when you're looking at your own code for improvement. Don't consider it an exhaustive list of code style guidelines, however. Over the course of the semester, we'll talk about a lot more things — specifications, abstract data types with representation invariants, concurrency and thread safety — which will then become fodder for code review.

Smelly Example #1

Programmers often describe bad code as having a “bad smell” that needs to be removed. “Code hygiene” is another word for this. Let's start with some smelly code.

```

public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth;
}

```

The next few sections and exercises will pick out the particular smells in this code example.

Don't Repeat Yourself

Duplicated code is a risk to safety. If you have identical or very similar code in two places, then the fundamental risk is that there's a bug in both copies, and some maintainer fixes the bug in one place but not the other.

Avoid duplication like you'd avoid crossing the street without looking. Copy-and-paste is an enormously tempting programming tool, and you should feel a frisson of danger run down your spine every time you use it. The longer the block you're copying, the riskier it is.

Don't Repeat Yourself ([//en.wikipedia.org/wiki/Don't_repeat_yourself](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself)) , or DRY for short, has become a programmer's mantra.

The `dayOfYear` example is full of identical code. How would you DRY it out?

Comments Where Needed

A quick general word about commenting. Good software developers write comments in their code, and do it judiciously. Good comments should make the code easier to understand, safer from bugs (because important assumptions have been documented), and ready for change.

One kind of crucial comment is a specification, which appears above a method or above a class and documents the behavior of the method or class. In Java, this is conventionally written as a Javadoc comment, meaning that it starts with `/**` and includes `@`-syntax, like `@param` and `@return` for methods. Here's an example of a spec:

```
/**
 * Compute the hailstone sequence.
 * See http://en.wikipedia.org/wiki/Collatz\_conjecture#Statement\_of\_the\_problem
 * @param n starting number of sequence; requires n > 0.
 * @return the hailstone sequence starting at n and ending with 1.
 *         For example, hailstone(3)=[3,10,5,16,8,4,2,1].
 */
public static List<Integer> hailstoneSequence(int n) {
    ...
}
```

Specifications document assumptions. We've already mentioned specs a few times, and there will be much more to say about them in a future reading.

Another crucial comment is one that specifies the provenance or source of a piece of code that was copied or adapted from elsewhere. This is vitally important for practicing software developers, and is required by the 6.005 collaboration policy ([..../general/collaboration.html](#)) when you adapt code you found on the web. Here is an example:

```
// read a web page into a string
// see http://stackoverflow.com/questions/4328711/read-url-to-string-in-few-lines-of-java-code
String mitHomepage = new Scanner(new URL("http://www.mit.edu").openStream(), "UTF-8").useDelimiter("\n").next();
```

One reason for documenting sources is to avoid violations of copyright. Small snippets of code on Stack Overflow are typically in the public domain, but code copied from other sources may be proprietary or covered by other kinds of open source licenses, which are more restrictive. Another reason for documenting sources is that the code can fall out of date; the Stack Overflow answer ([//stackoverflow.com/questions/4328711/read-url-to-string-in-few-lines-of-java-code](#)) from which this code came has evolved significantly in the years since it was first answered.

Some comments are bad and unnecessary. Direct transliterations of code into English, for example, do nothing to improve understanding, because you should assume that your reader at least knows Java:

```
while (n != 1) { // test whether n is 1   (don't write comments like this!)
    ++i; // increment i
    l.add(n); // add n to l
}
```

But obscure code should get a comment:

```
sendMessage("as you wish"); // this basically says "I love you"
```

The `dayOfYear` code needs some comments — where would you put them? For example, where would you document whether `month` runs from 0 to 11 or from 1 to 12?

Fail Fast

Failing fast means that code should reveal its bugs as early as possible. The earlier a problem is observed (the closer to its cause), the easier it is to find and fix. As we saw in the first reading ([..01-static-checking/#static_checking_dynamic_checking_no_checking](#)), static checking fails faster than dynamic checking, and dynamic checking fails faster than producing a wrong answer that may corrupt subsequent computation.

The `day0fYear` function doesn't fail fast — if you pass it the arguments in the wrong order, it will quietly return the wrong answer. In fact, the way `day0fYear` is designed, it's highly likely that a non-American will pass the arguments in the wrong order! It needs more checking — either static checking or dynamic checking.

Avoid Magic Numbers

There are really only two constants that computer scientists recognize as valid in and of themselves: 0, 1, and maybe 2. (Okay, three constants.)

All other constants are called magic

([https://en.wikipedia.org/wiki/Magic_number_\(programming\)#Unnamed_numerical_constants](https://en.wikipedia.org/wiki/Magic_number_(programming)#Unnamed_numerical_constants)) because they appear as if out of thin air with no explanation.

One way to explain a number is with a comment, but a far better way is to declare the number as a named constant with a good, clear name.

`day0fYear` is full of magic numbers:

- The months 2, ..., 12 would be far more readable as `FEBRUARY`, ..., `DECEMBER`.
- The days-of-months 30, 31, 28 would be more readable (and eliminate duplicate code) if they were in a data structure like an array, list, or map, e.g. `MONTH_LENGTH[month]`.
- The mysterious numbers 59 and 90 are particularly pernicious examples of magic numbers. Not only are they uncommented and undocumented, they are actually the result of a *computation done by hand* by the programmer. Don't hardcode constants that you've computed by hand. Java is better at arithmetic than you are. Explicit computations like `31 + 28` make the provenance of these mysterious numbers much clearer. `MONTH_LENGTH[JANUARY] + MONTH_LENGTH[FEBRUARY]` would be clearer still.

One Purpose For Each Variable

In the `day0fYear` example, the parameter `day0fMonth` is reused to compute a very different value — the return value of the function, which is not the day of the month.

Don't reuse parameters, and don't reuse variables. Variables are not a scarce resource in programming. Introduce them freely, give them good names, and just stop using them when you stop needing them. You will confuse your reader if a variable that used to mean one thing suddenly starts meaning something different a few lines down.

Not only is this an ease-of-understanding question, but it's also a safety-from-bugs and ready-for-change question.

Method parameters, in particular, should generally be left unmodified. (This is important for being ready-for-change — in the future, some other part of the method may want to know what the original parameters of the method were, so you shouldn't blow them away while you're computing.) It's a good idea to use `final` for method parameters, and as many other variables as you can. The `final` keyword says that the variable should never be reassigned, and the Java compiler will check it statically. For example:

```
public static int day0fYear(final int month, final int day0fMonth, final int year) {  
    ...  
}
```

Smelly Example #2

There was a latent bug in `day0fYear`. It didn't handle leap years at all. As part of fixing that, suppose we write a leap-year method.

```
public static boolean leap(int y) {
    String tmp = String.valueOf(y);
    if (tmp.charAt(2) == '1' || tmp.charAt(2) == '3' || tmp.charAt(2) == 5 || tmp.charAt(2) == '7' || tmp.charAt(2) == '9') {
        if (tmp.charAt(3)=='2'||tmp.charAt(3)=='6') return true; /*R1*/
        else
            return false; /*R2*/
    }else{
        if (tmp.charAt(2) == '0' && tmp.charAt(3) == '0') {
            return false; /*R3*/
        }
        if (tmp.charAt(3)=='0'||tmp.charAt(3)=='4'||tmp.charAt(3)=='8')return true; /*R4*/
    }
    return false; /*R5*/
}
```

What are the bugs hidden in this code? And what style problems that we've already talked about?

Use Good Names

Good method and variable names are long and self-descriptive. Comments can often be avoided entirely by making the code itself more readable, with better names that describe the methods and variables.

For example, you can rewrite

```
int tmp = 86400; // tmp is the number of seconds in a day (don't do this!)
```

as:

```
int secondsPerDay = 86400;
```

In general, variable names like `tmp`, `temp`, and `data` are awful, symptoms of extreme programmer laziness. Every local variable is temporary, and every variable is data, so those names are generally meaningless. Better to use a longer, more descriptive name, so that your code reads clearly all by itself.

Follow the lexical naming conventions of the language. In Python, classes are typically Capitalized, variables are lowercase, and words_are_separated_by_underscores. In Java:

- `methodsAreNamedWithCamelCaseLikeThis`
- `variablesAreAlsoCamelCase`
- `CONSTANTS_ARE_IN_ALL_CAPS_WITH_UNDERSCORES`
- `ClassesAreCapitalized`
- `packages.are.lowercase.and.separated.by.dots`

Method names are usually verb phrases, like `getDate` or `isUpperCase`, while variable and class names are usually noun phrases. Choose short words, and be concise, but avoid abbreviations. For example, `message` is clearer than `msg`, and `word` is so much better than `wd`. Keep in mind that many of your teammates in class and in the real world will not be native English speakers, and abbreviations can be even harder for non-native speakers.

The `leap` method has bad names: the method name itself, and the local variable name. What would you call them instead?

Use Whitespace to Help the Reader

Use consistent indentation. The `leap` example is bad at this. The `day0fYear` example is much better. In fact, `day0fYear` nicely lines up all the numbers into columns, making them easy for a human reader to compare and check. That's a great use of whitespace.

Put spaces within code lines to make them easy to read. The `leap` example has some lines that are packed together — put in some spaces.

Never use tab characters for indentation, only space characters. Note that we say *characters*, not keys. We're not saying you should never press the Tab key, only that your editor should never put a tab character into your source file in response to your pressing the Tab key. The reason for this rule is that different tools treat tab characters differently — sometimes expanding them to 4 spaces, sometimes to 2 spaces, sometimes to 8. If you run “git diff” on the command line, or if you view your source code in a different editor, then the indentation may be completely screwed up. Just use spaces. Always set your programming editor to insert space characters when you press the Tab key.

Smelly Example #3

Here's a third example of smelly code that will illustrate the remaining points of this reading.

```
public static int LONG_WORD_LENGTH = 5;
public static String longestWord;

public static void countLongWords(List<String> words) {
    int n = 0;
    longestWord = "";
    for (String word: words) {
        if (word.length() > LONG_WORD_LENGTH) ++n;
        if (word.length() > longestWord.length()) longestWord = word;
    }
    System.out.println(n);
}
```

Don't Use Global Variables

Avoid global variables. Let's break down what we mean by *global variable*. A global variable is:

- a *variable*, a name whose meaning can be changed
- that is *global*, accessible and changeable from anywhere in the program.

Why Global Variables Are Bad ([//c2.com/cgi/wiki?GlobalVariablesAreBad](http://c2.com/cgi/wiki?GlobalVariablesAreBad)) has a good list of the dangers of global variables.

In Java, a global variable is declared `public static`. The `public` modifier makes it accessible anywhere, and `static` means there is a single instance of the variable.

In general, change global variables into parameters and return values, or put them inside objects that you're calling methods on. We'll see many techniques for doing that in future readings.

Methods Should Return Results, not Print Them

`countLongWords` isn't ready for change. It sends some of its result to the console, `System.out`. That means that if you want to use it in another context — where the number is needed for some other purpose, like computation rather than human eyes — it would have to be rewritten.

In general, only the highest-level parts of a program should interact with the human user or the console. Lower-level parts should take their input as parameters and return their output as results. The sole exception here is debugging output, which can of course be printed to the console. But that kind of output shouldn't be a part of your design, only a part of how you debug your design.

Summary

Code review is a widely-used technique for improving software quality by human inspection. Code review can detect many kinds of problems in code, but as a starter, this reading talked about these general principles of good code:

- Don't Repeat Yourself (DRY)
- Comments where needed
- Fail fast
- Avoid magic numbers
- One purpose for each variable
- Use good names
- No global variables
- Return results, don't print them
- Use whitespace for readability

The topics of today's reading connect to our three key properties of good software as follows:

- **Safe from bugs.** In general, code review uses human reviewers to find bugs. DRY code lets you fix a bug in only one place, without fear that it has propagated elsewhere. Commenting your assumptions clearly makes it less likely that another programmer will introduce a bug. The Fail Fast principle detects bugs as early as possible. Avoiding global variables makes it easier to localize bugs related to variable values, since non-global variables can be changed in only limited places in the code.
- **Easy to understand.** Code review is really the only way to find obscure or confusing code, because other people are reading it and trying to understand it. Using judicious comments, avoiding magic numbers, keeping one purpose for each variable, using good names, and using whitespace well can all improve the understandability of code.
- **Ready for change.** Code review helps here when it's done by experienced software developers who can anticipate what might change and suggest ways to guard against it. DRY code is more ready for change, because a change only needs to be made in one place. Returning results instead of printing them makes it easier to adapt the code to a new purpose.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 5: Version Control

Introduction

Inventing version control

Git

Copy an object graph with git clone

What else is in the object graph?

Add to the object graph with git commit

Send & receive object graphs with git push & git pull

Merging

Why do commits look like diffs?

Version control and the big three

Reading 5: Version Control

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

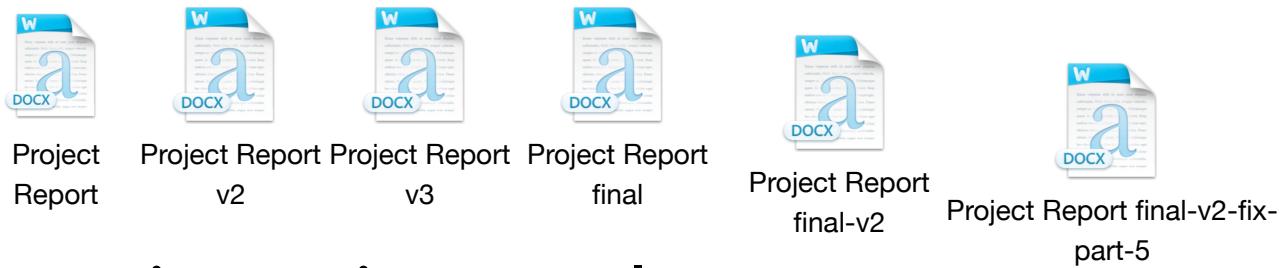
- Know what version control is and why we use it
- Understand how Git stores version history as a graph
- Practice reading, creating, and using version history

Introduction

Version control systems ([//en.wikipedia.org/wiki/Revision_control](https://en.wikipedia.org/wiki/Revision_control)) are essential tools of the software engineering world. More or less every project — serious or hobby, open source or proprietary — uses version control. Without version control, coordinating a team of programmers all editing the same project's code will reach pull-out-your-hair levels of aggravation.

Version control systems you've already used

- Dropbox
- Undo/redo buffer ([//en.wikipedia.org/wiki/Undo](https://en.wikipedia.org/wiki/Undo))
- Keeping multiple copies of files with version numbers

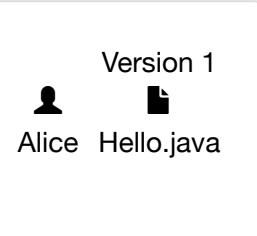


Inventing version control

Suppose Alice ([//en.wikipedia.org/wiki/Alice_and_Bob](https://en.wikipedia.org/wiki/Alice_and_Bob)) is working on a problem set by herself.

She starts with one file `Hello.java` in her pset, which she works on for several days.

At the last minute before she needs to hand in her pset to be graded, she realizes she has made a change that breaks everything. If only she could go back in time and retrieve a past version!



A simple discipline of saving backup files would get the job done.

Alice uses her judgment to decide when she has reached some milestone that justifies saving the code. She saves the versions of `Hello.java` as `Hello.1.java`, `Hello.2.java`, and `Hello.java`. She follows the convention that the most recent version is just `Hello.java` to avoid confusing Eclipse. We will call the most recent version the *head*.

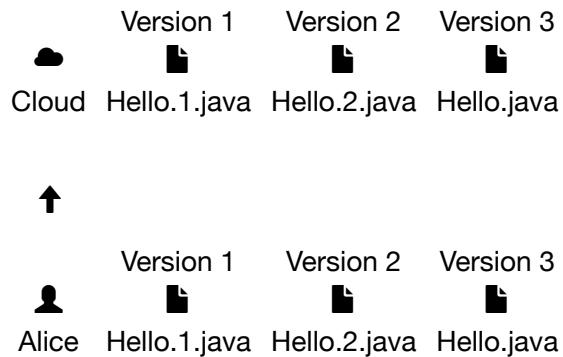


Now when Alice realizes that version 3 is fatally flawed, she can just copy version 2 back into the location for her current code. Disaster averted! But what if version 3 included some changes that were good and some that were bad? Alice can compare the files manually to find the changes, and sort them into good and bad changes. Then she can copy the good changes into version 2.

This is a lot of work, and it's easy for the human eye to miss changes. Luckily, there are standard software tools for comparing text; in the UNIX world, one such tool is `diff` ([/en.wikipedia.org/wiki/Diff](https://en.wikipedia.org/wiki/Diff)). A better version control system will make diffs easy to generate.

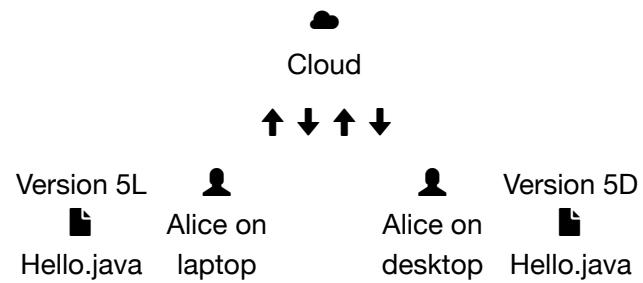
Alice also wants to be prepared in case her laptop gets run over by a bus, so she saves a backup of her work in the cloud, uploading the contents of her working directory whenever she's satisfied with its contents.

If her laptop is kicked into the Charles, Alice can retrieve the backup and resume work on the pset on a fresh machine, retaining the ability to time-travel back to old versions at will.



Furthermore, she can develop her pset on multiple machines, using the cloud provider as a common interchange point. Alice makes some changes on her laptop and uploads them to the cloud. Then she downloads onto her desktop machine at home, does some more work, and uploads the improved code (complete with old file versions) back to the cloud.

If Alice isn't careful, though, she can run into trouble with this approach. Imagine that she starts editing `Hello.java` to create "version 5" on her laptop. Then she gets distracted and forgets about her changes. Later, she starts working on a new "version 5" on her desktop machine, including *different* improvements. We'll call these versions "5L" and "5D," for "laptop" and "desktop."



When it comes time to upload changes to the cloud, there is an opportunity for a mishap! Alice might copy all her local files into the cloud, causing it to contain version 5D only. Later Alice syncs from the cloud to her laptop, potentially overwriting version 5L, losing the worthwhile changes. What Alice really wants here is a *merge*, to create a new version based on the two version 5's.

At this point, considering just the scenario of one programmer working alone, we already have a list of operations that should be supported by a version control scheme:

- *reverting* to a past version
- *comparing* two different versions
- *pushing* full version history to another location
- *pulling* history back from that location
- *merging* versions that are offshoots of the same earlier version

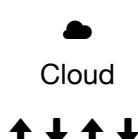
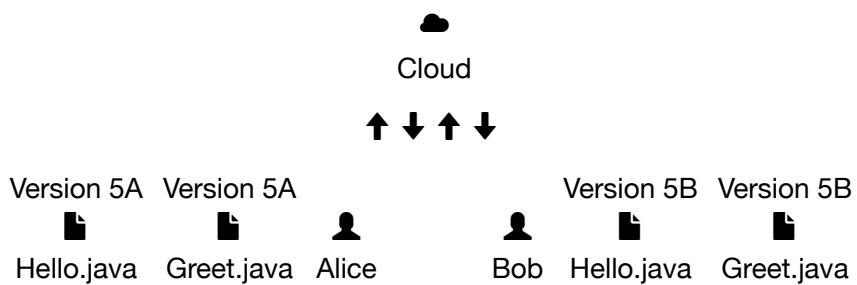
Multiple developers

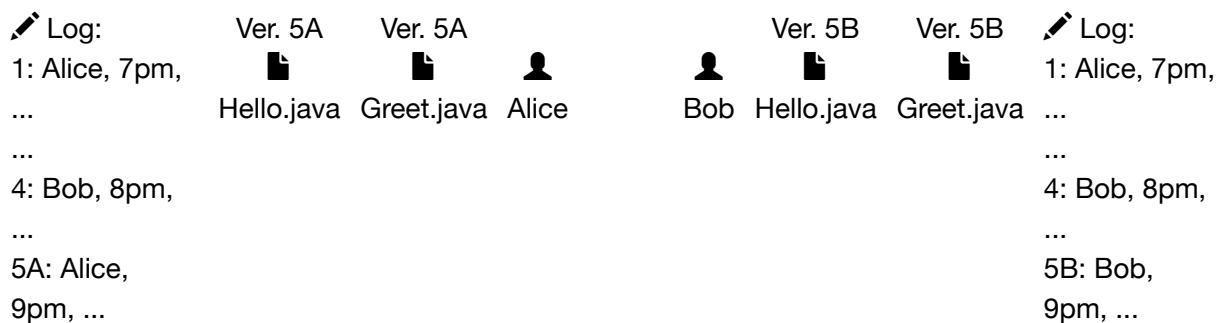
Now let's add into the picture Bob, another developer. The picture isn't too different from what we were just thinking about.

Alice and Bob here are like the two Alices working on different computers. They no longer share a brain, which makes it even more important to follow a strict discipline in pushing to and pulling from the shared cloud server. The two programmers must coordinate on a scheme for coming up with

version numbers. Ideally, the scheme allows us to assign clear names to *whole sets of files*, not just individual files. (Files depend on other files, so thinking about them in isolation allows inconsistencies.)

Merely uploading new source files is not a very good way to communicate to others the high-level idea of a set of changes. So let's add a log that records for each version *who* wrote it, *when* it was finalized, and *what* the changes were, in the form of a short human-authored message.





Pushing another version now gets a bit more complicated, as we need to merge the logs. This is easier to do than for Java files, since logs have a simpler structure – but without tool support, Alice and Bob will need to do it manually! We also want to enforce consistency between the logs and the actual sets of available files: for each log entry, it should be easy to extract the complete set of files that were current at the time the entry was made.

But with logs, all sorts of useful operations are enabled. We can look at the log for just a particular file: a view of the log restricted to those changes that involved modifying some file. We can also use the log to figure out which change contributed each line of code, or, even better, which person contributed each line, so we know who to complain to when the code doesn't work. This sort of operation would be tedious to do manually; the automated operation in version control systems is called *annotate* (or, unfortunately, *blame*).

Multiple branches

It sometimes makes sense for a subset of the developers to go off and work on a *branch*, a parallel code universe for, say, experimenting with a new feature. The other developers don't want to pull in the new feature until it is done, even if several coordinated versions are created in the meantime. Even a single developer can find it useful to create a branch, for the same reasons that Alice was originally using the cloud server despite working alone.

In general, it will be useful to have many shared places for exchanging project state. There may be multiple branch locations at once, each shared by several programmers. With the right set-up, any programmer can pull from or push to any location, creating serious flexibility in cooperation patterns.

The shocking conclusion

Of course, it turns out we haven't invented anything here: Git ([//git-scm.com](http://git-scm.com)) does all these things for you, and so do many other version control systems.

Distributed vs. centralized

Traditional *centralized* version control systems like CVS and Subversion ([//subversion.apache.org/](http://subversion.apache.org/)) do a subset of the things we've imagined above. They support a collaboration graph – who's sharing what changes with whom – with one master server and copies that only communicate with the master.

In a centralized system, everyone must share their work to and from the master repository. Changes are safely stored *in version control* if they are *in the master repository*, because that's the only repository.

Dan Carol



Cloud



Alice Bob

Dan Carol



Cloud



Alice Bob

In contrast, *distributed* version control systems like Git ([//git-scm.com](http://git-scm.com)) and Mercurial (<https://www.mercurial-scm.org/>) allow all sorts of different collaboration graphs, where teams and subsets of teams can experiment easily with alternate versions of code and history, merging versions together as they are determined to be good ideas.

In a distributed system, all repositories are created equal, and it's up to users to assign them different roles. Different users might share their work to and from different repos, and the team must decide what it means for a change to be *in version control*. If the change is stored in just a single programmer's repo, do they still need to share it with a designated collaborator or specific server before the rest of the team considers it official?

Version control terminology

- **Repository** : a local or remote store of the versions in our project
- **Working copy** : a local, editable copy of our project that we can work on
- **File** : a single file in our project
- **Version or revision** : a record of the contents of our project at a point in time
- **Change or diff** : the difference between two versions
- **Head** : the current version

Features of a version control system

- **Reliable** : keep versions around for as long as we need them; allow backups
- **Multiple files** : track versions of a project, not single files
- **Meaningful versions** : what were the changes, why where they made?
- **Revert** : restore old versions, in whole or in part
- **Compare versions**
- **Review history** : for the whole project or individual files
- **Not just for code** : prose, images, ...

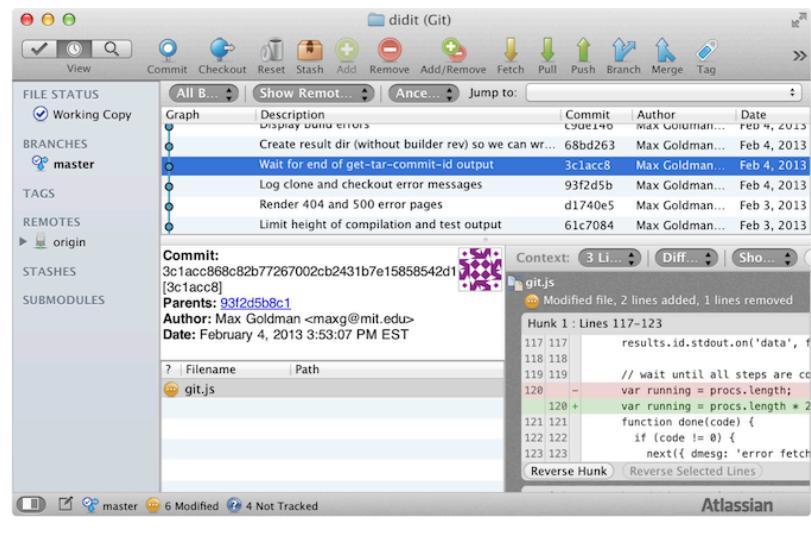
It should **allow multiple people to work together** :

- **Merge** : combine versions that diverged from a common previous version
- **Track responsibility** : who made that change, who touched that line of code?
- **Work in parallel** : allow one programmer to work on their own for a while (without giving up version control)
- **Work-in-progress** : allow multiple programmers to share unfinished work (without disrupting others, without giving up version control)

Git

The version control system we'll use in 6.005 is Git ([//git-scm.com](http://git-scm.com)) . It's powerful and worth learning. But Git's user interface can be terribly frustrating. What is Git's user interface?

- In 6.005, we will use Git on the command line. The command line is a fact of life, ubiquitous because it is so powerful.
- The command line can make it very difficult to see what is going on in your repositories. You may find SourceTree ([//www.sourcetreeapp.com](http://www.sourcetreeapp.com)) (shown on the right) for Mac & Windows useful. On any platform, gitk ([//git-scm.com/docs/gitk](http://git-scm.com/docs/gitk)) can give you a basic Git GUI. Ask Google for other suggestions.



An important note about tools for Git:

- Eclipse has built-in support for Git. If you follow the problem set instructions ([..../psets/ps0/](#)), Eclipse will know your project is in Git and will show you helpful icons. We do not recommend using the Eclipse Git UI to make changes, commit, etc., and course staff may not be able to help you with problems.
- GitHub ([//github.com/](http://github.com/)) makes desktop apps for Mac and Windows. Because the GitHub app changes how some Git operations work, if you use the GitHub app, course staff will not be able to help you.

Getting started with Git

On the Git ([//git-scm.com](http://git-scm.com)) website, you can find two particularly useful resources:

- *Pro Git* ([//git-scm.com/book](http://git-scm.com/book)) documents everything you might need to know about Git.
- The Git command reference ([//git-scm.com/docs](http://git-scm.com/docs)) can help with the syntax of Git commands.

You've already completed **PS0** ([..../psets/ps0/#clone](#)) and the **Getting Started intro to Git** ([..../getting-started/#git](#)) .

The Git object graph

Read: **Pro Git 1.3: Git Basics** ([//git-scm.com/book/en/v2/Getting-Started-Git-Basics](http://git-scm.com/book/en/v2/Getting-Started-Git-Basics))

That reading introduces the three pieces of a Git repo: `.git` directory, working directory, and staging area.

All of the operations we do with Git — clone, add, commit, push, log, merge, ... — are operations on a graph data structure that stores all of the versions of files in our project, and all the log entries describing those changes. The **Git object graph** is stored in the `.git` directory of your local repository. Another copy of the graph, e.g. for PS0, is on Athena in:

`/mit/6.005/git/sp16/psets/ps0/[your username].git`

Copy an object graph with `git clone`

How do you get the object graph from Athena to your local machine in order to start working on the problem set? `git clone` copies the graph.

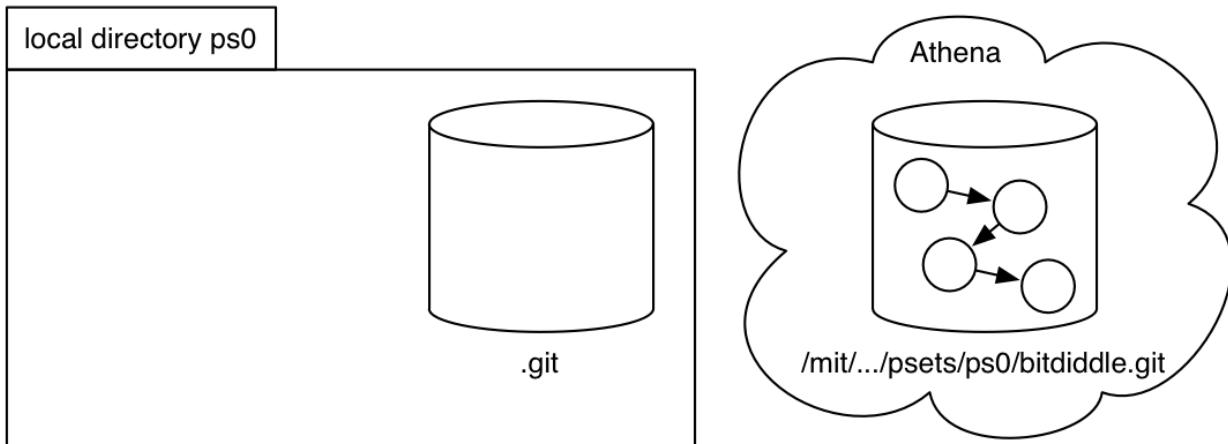
Suppose your username is `bitdiddle` :

```
git clone ssh://.../psets/ps0/bitdiddle.git ps0
```

Hover or tap on each step to update the diagram below:

1. Create an empty local directory `ps0` , and `ps0/.git` .
2. Copy the object graph from `ssh://.../psets/ps0/bitdiddle.git` into `ps0/.git` .
3. **Check out** the current version of the `master` branch .

Diagram for highlighted step:



We still haven't explained what's in the object graph. But before we do that, let's understand step 3 of `git clone` : check out the current version of the `master` branch.

The object graph is stored on disk in a convenient and efficient structure for performing Git operations, but not in a format we can easily use. In Alice's invented version control scheme , the current version of `Hello.java` was just called `Hello.java` because she needed to be able to edit it normally. In Git, we obtain normal copies of our files by *checking them out* from the object graph. These are the files we see and edit in Eclipse.

We also decided above that it might be useful to support multiple *branches* in the version history . Multiple branches are essential for large teams working on long-term projects. To keep things simple in 6.005, we will not use branches and we don't recommend that you create any. Every Git repo comes with a default branch called `master` , and all of our work will be on the `master` branch.

So step 2 of `git clone` gets us an object graph, and step 3 gets us a **working directory** full of files we can edit, starting from the current version of the project.

Let's finally dive into that object graph!

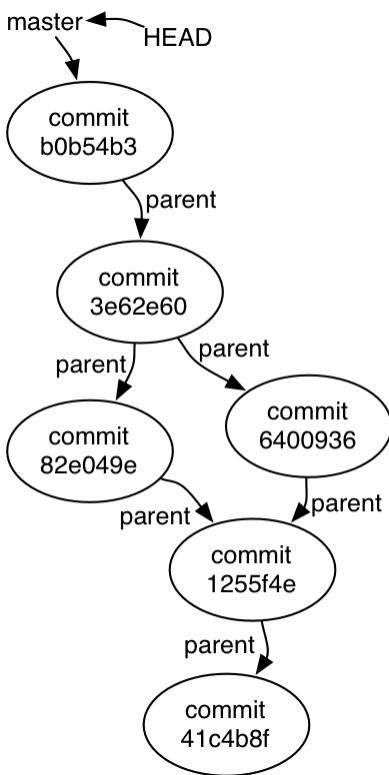
Clone an example repo: <https://github.com/mit6005/sp16-ex05-hello-git.git>

Using commands from **Getting Started** ([../../getting-started/#git](#)) or **Pro Git 2.3: Viewing the Commit History** ([//git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History](#)) , or by using a tool like SourceTree, explain the history of this little project to yourself.

Here's the output of `git log` (./../getting-started/#config-git) for this example repository:

```
* b0b54b3 (HEAD, origin/master, origin/HEAD, master) Greeting in Java
* 3e62e60 Merge
|\
| * 6400936 Greeting in Scheme
* | 82e049e Greeting in Ruby
|/
* 1255f4e Change the greeting
* 41c4b8f Initial commit
```

The history of a Git project is a **directed acyclic graph** ([//en.wikipedia.org/wiki/Directed_acyclic_graph](https://en.wikipedia.org/wiki/Directed_acyclic_graph)) (DAG). The history graph is the backbone of the full object graph stored in `.git`, so let's focus on it for a minute.



Each node in the history graph is a **commit** a.k.a. **version** a.k.a. **revision** of the project: a complete snapshot of all the files in the project at that point in time. You may recall from our earlier reading (./../getting-started/#getting_the_history_of_the_repository) that each commit is identified by a unique ID, displayed as a hexadecimal number.

Except for the initial commit, each commit has a pointer to its **parent** commit. For example, commit `1255f4e` has parent `41c4b8f` : this means `41c4b8f` happened first, then `1255f4e`.

Some commits have the same parent: they are versions that diverged from a common previous version. And some commits have two parents: they are versions that tie divergent histories back together.

A branch — remember `master` will be our only branch for now — is just a name that points to a commit.

Finally, HEAD points to our current commit — almost. We also need to remember which branch we're working on. So HEAD points to the current branch, which points to the current commit.

Check your understanding...

What else is in the object graph?

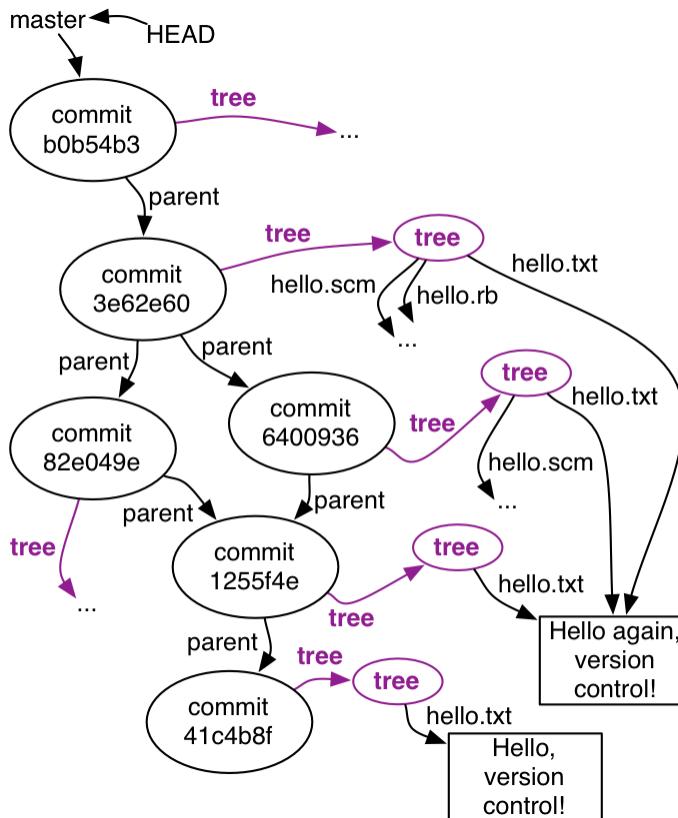
The history graph is the backbone of the full object graph. What else is in there?

Each commit is a snapshot of our entire project, which Git represents with a **tree** node. For a project of any reasonable size, most of the files *won't* change in any given revision. Storing redundant copies of the files would be wasteful, so Git doesn't do that.

Instead, the Git object graph stores each version of an individual file *once*, and allows multiple commits to *share* that one copy. To the left is a more complete rendering of the Git object graph for our example.

Keep this picture in the back of your mind, because it's a wonderful example of the sharing enabled by *immutable data types*, which we're going to discuss a few classes from now.

Each commit also has log data — who, when, short log message, etc. — not shown in the diagram.



Add to the object graph with `git commit`

How do we add new commits to the history graph? `git commit` creates a new commit.

In some alternate universe, `git commit` might create a new commit based on the current contents of your working directory. So if you edited `Hello.java` and then did `git commit`, the snapshot would include your changes.

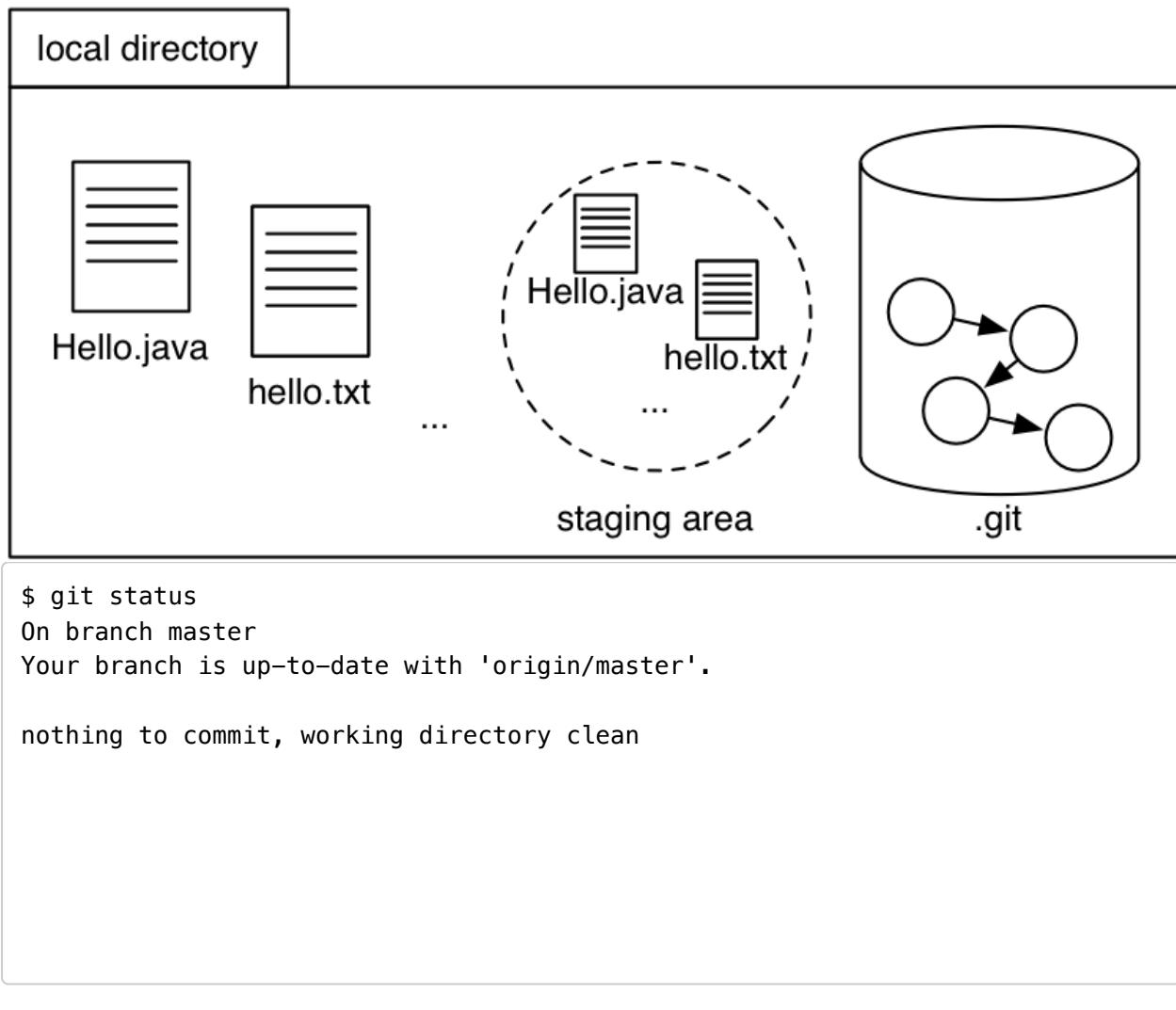
We're not in that universe; in our universe, Git uses that third and final piece of the repository: the **staging area** (a.k.a. the **index**, which is only a useful name to know because sometimes it shows up in documentation).

The staging area is like a proto-commit, a commit-in-progress. Here's how we use the staging area and `git add` to build up a new snapshot, which we then cast in stone using `git commit`:

```
Modify hello.txt , git add hello.txt , git commit
```

Hover or tap on each step to update the diagram, and to see the output of `git status` at each step:

1. If we haven't made any changes yet, then the working directory, staging area, and HEAD commit are all identical.
2. Make a change to a file. For example, let's edit `hello.txt`. Other changes might be creating a new file, or deleting a file.
3. **Stage** those changes using `git add`.
4. Create a new commit out of all the staged changes using `git commit`.



```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean
```

Use `git status` frequently to keep track of whether you have no changes, unstaged changes, or staged changes; and whether you have new commits in your local repository that haven't been pushed.

Sequences, trees, and graphs

When you're working independently, on a single machine, the DAG of your version history will usually look like a sequence: commit 1 is the parent of commit 2 is the parent of commit 3...

There are three programmers involved in the history of our example repository. Two of them – Alyssa and Ben – made changes “at the same time.” In this case, “at the same time” doesn’t mean precisely contemporaneous. Instead, it means they made two different *new* versions based on the same *previous* version, just as Alice made version 5L and 5D on her laptop and desktop .

When multiple commits share the same parent commit, our history DAG changes from a sequence to a tree: it branches apart. Notice that a branch in the history of the project doesn’t require anyone to create a new Git branch, merely that we start from the same commit and work in parallel on different copies of the repository:

```
:
* commit 82e049e248c63289b8a935ce71b130a74dc04152
| Author: Ben Bitdiddle <ben.bitdiddle@example.com>
| Greeting in Ruby
|
| * commit 64009369c5ab93492931ad07962ee81bda921ded
|/ Author: Alyssa P. Hacker <alyssa.p.hacker@example.com>
| Greeting in Scheme
|
* commit 1255f4e4a5836501c022deb337fd3f8800b02e4
| Author: Max Goldman <maxg@mit.edu>
| Change the greeting
:
```

Finally, the history DAG changes from tree- to graph-shaped when the branching changes are merged together:

```
:
* commit 3e62e60a7b4a0c262cd8eb4308ac3e5a1e94d839
|\ Author: Max Goldman <maxg@mit.edu>
| | Merge
| |
* | commit 82e049e248c63289b8a935ce71b130a74dc04152
| | Author: Ben Bitdiddle <ben.bitdiddle@example.com>
| | Greeting in Ruby
| |
| * commit 64009369c5ab93492931ad07962ee81bda921ded
|/ Author: Alyssa P. Hacker <alyssa.p.hacker@example.com>
| Greeting in Scheme
|
* commit 1255f4e4a5836501c022deb337fd3f8800b02e4
| Author: Max Goldman <maxg@mit.edu>
| Change the greeting
:
```

How is it that changes are merged together? First we'll need to understand how history is shared between different users and repositories.

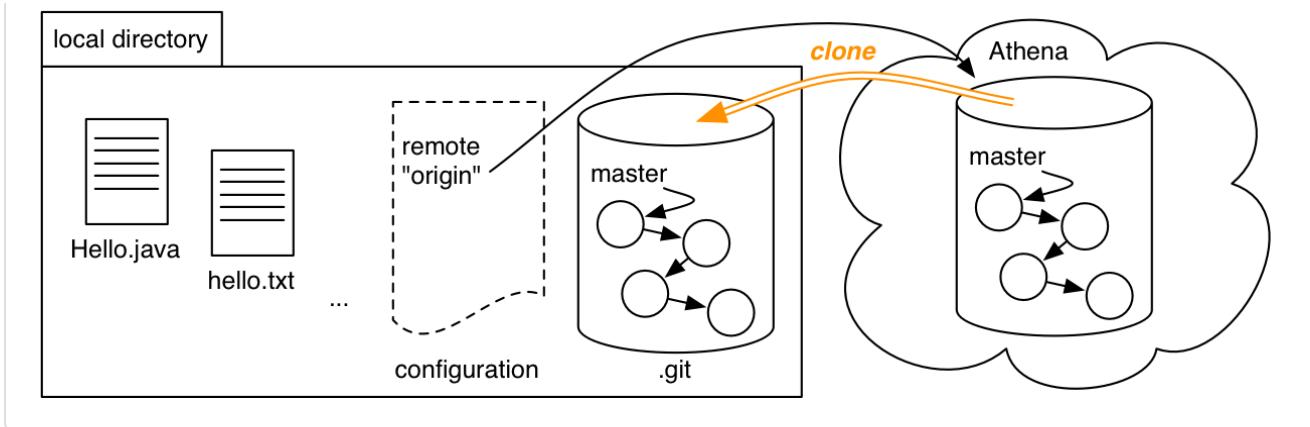
Send & receive object graphs with `git push` & `git pull`

We can send new commits to a remote repository using `git push` :

```
git push origin master
```

Hover or tap on each step to update the diagram:

1. When we clone a repository, we obtain a copy of the history graph.
Git remembers where we cloned from as a **remote repository** called `origin` .
2. Using `git commit` , we add new commits to the local history on the `master` branch.
3. To send those changes back to the `origin` remote, use `git push origin master` .



And we receive new commits using `git pull`. Note that `git pull`, in addition to fetching new parts of the object graph, also updates the working copy by checking out the latest version (just like `git clone` checked out a working copy to start with).

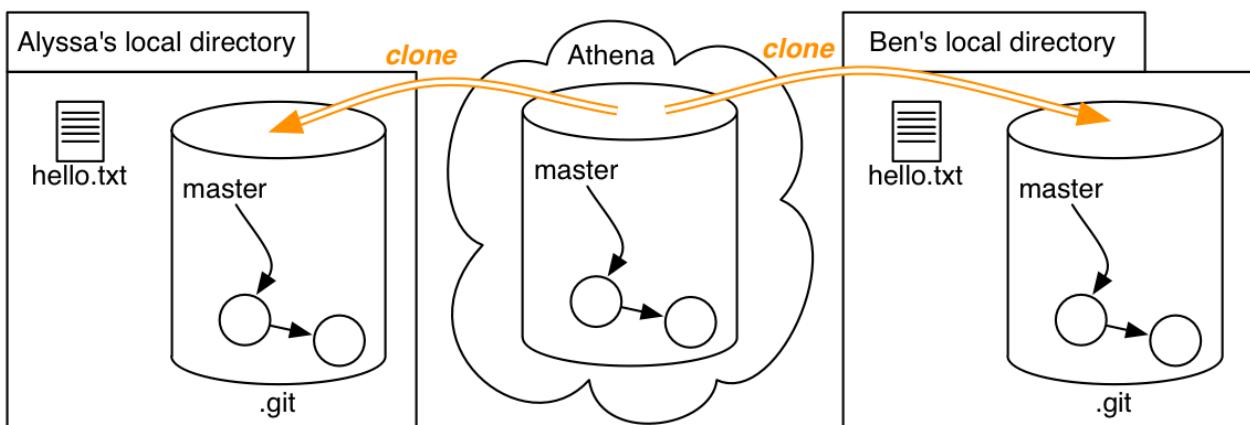
Merging

Now, let's examine what happens when changes occur in parallel:

Create and commit `hello.scm` and `hello.rb` in parallel

Hover or tap on each step to update the diagram:

1. Both Alyssa and Ben **clone** the repository with two commits (`41c4b8f` and `1255f4e`).
2. Alyssa creates `hello.scm` and **commits** her change as `6400936` .
3. At the same time, Ben creates `hello.rb` and **commits** his change as `82e049e` .
At this point, both of their changes only exist in their local repositories. In each repo, `master` now points to a different commit.
4. Let's suppose Alyssa is the first to **push** her change up to Athena.
5. What happens if Ben tries to push now? The push will be rejected: if the server updates `master` to point to Ben's commit, Alyssa's commit will disappear from the project history!
6. Ben must **merge** his changes with Alyssa's.
To perform the merge, he **pulls** her commit from Athena, which does two things:
 - (a) Downloads new commits into Ben's repository's object graph
7. (b) Merges Ben's history with Alyssa's, creating a new commit (`3e62e60`) that joins together the disparate histories. This commit is a snapshot like any other: a snapshot of the repository with both of their changes applied.
8. Now Ben can `git push`, because no history will go missing when he does.
9. And Alyssa can `git pull` to obtain Ben's work.



In this example, Git was able to merge Alyssa's and Ben's changes automatically, because they each modified different files. If both of them had edited the *same parts of the same files*, Git would report a **merge conflict**. Ben would have to manually weave their changes together before committing the merge. All of this is discussed in the Getting Started section on merges, merging, and merge conflicts (./../getting-started/#merges).

Why do commits look like diffs?

We've defined a commit as a snapshot of our entire project, but if you ask Git, it doesn't seem to see things that way:

```
$ git show 1255f4e
commit 1255f4e4a5836501c022deb337fd3f8800b02e4
Author: Max Goldman <maxg@mit.edu>
Date:   Mon Sep 14 14:58:40 2015 -0400

    Change the greeting

diff --git a/hello.txt b/hello.txt
index c1106ab..3462165 100644
--- a/hello.txt
+++ b/hello.txt
@@ -1 +1 @@
-Hello, version control!
+Hello again, version control!
```

Git is assuming that most of our project does not change in any given commit, so showing only the differences will be more useful. Almost all the time, that's true.

But we can ask Git to show us what was in the repo at a particular commit:

```
$ git show 3e62e60:
tree 3e62e60:

hello.rb
hello.scm
hello.txt
```

Yes, the addition of a : completely changes the meaning of that command.

We can also see what was in a particular file in that commit:

```
$ git show 3e62e60:hello.scm
(display "Hello, version control!")
```

This is one of the simplest ways you can use Git to recover from a disaster: ask it to `git show` you the contents of a now-broken file at some earlier version when the file was OK.

We'll practice some disaster recovery commands in class.

Version control and the big three

How does version control relate to the three big ideas of 6.005?

Safe from bugs find when and where something broke
look for other, similar mistakes
gain confidence that code hasn't changed accidentally

Easy to understand why was a change made?
what else was changed at the same time?
who can I ask about this code?

Ready for change all about managing and organizing changes
accept and integrate changes from other developers
isolate speculative work on branches

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 6: Specifications

Introduction

Part 1: Specifications

Part 2: Exceptions

Summary

Reading 6: Specifications

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- Understand preconditions and postconditions in method specifications, and be able to write correct specifications
- Be able to write tests against a specification
- Know the difference between checked and unchecked exceptions in Java
- Understand how to use exceptions for special results

Introduction

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 precondition, the client has responsibilities too.

In this reading we'll look at the role played by specifications of methods. We'll discuss what preconditions and postconditions are, and what they mean for the implementor and the client of a method. We'll also talk about how to use exceptions, an important language feature found in Java, Python, and many other modern languages, which allows us to make a method's interface safer from bugs and easier to understand.

Part 1: Specifications (specs/)

Part 2: Exceptions (exceptions/)

Summary

Before we wrap up, check your understanding with one last example:

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.

Let's review how specifications help with the main goals of this course:

- **Safe from bugs** . A good specification clearly documents the mutual assumptions that a client and implementor are relying on. Bugs often come from disagreements at the interfaces, and the presence of a specification reduces that. Using machine-checked language features in your spec, like static typing and exceptions rather than just a human-readable comment, can reduce bugs still more.
- **Easy to understand** . A short, simple spec is easier to understand than the implementation itself, and saves other people from having to read the code.
- **Ready for change** . Specs establish contracts between different parts of your code, allowing those parts to change independently as long as they continue to satisfy the requirements of the contract.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 7: Designing Specifications

Introduction

Deterministic vs. underdetermined specs

Declarative vs. operational specs

Stronger vs. weaker specs

Diagramming specifications

Designing good specifications

Precondition or postcondition?

About access control

About static vs. instance methods

Summary

Reading 7: Designing Specifications

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- Understand underdetermined specs, and be able to identify and assess nondeterminism
- Understand declarative vs. operational specs, and be able to write declarative specs
- Understand strength of preconditions, postconditions, and specs; and be able to compare spec strength
- Be able to write coherent, useful specifications of appropriate strength

Introduction

In this reading we'll look at different specs for similar behaviors, and talk about the tradeoffs between them. We'll look at three dimensions for comparing specs:

- How **deterministic** it is. Does the spec define only a single possible output for a given input, or allow the implementor to choose from a set of legal outputs?
- How **declarative** it is. Does the spec just characterize *what* the output should be, or does it explicitly say *how* to compute the output?
- How **strong** it is. Does the spec have a small set of legal implementations, or a large set?

Not all specifications we might choose for a module are equally useful, and we'll explore what makes some specifications better than others.

Deterministic vs. underdetermined specs

Recall the two example implementations of `find` we began with in the previous reading (./06-specifications) :

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

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

The subscripts `First` and `Last` are not actual Java syntax. We're using them here to distinguish the two implementations for the sake of discussion. In the actual code, both implementations should be Java methods called `find`.

Here is one possible specification of `find` :

```
static int findExactlyOne(int[] arr, int val)
    requires: val occurs exactly once in arr
    effects: returns index i such that arr[i] = val
```

This specification is **deterministic** : when presented with a state satisfying the precondition, the outcome is completely determined. Only one return value and one final state is possible. There are no valid inputs for which there is more than one valid output.

Both `findFirst` and `findLast` satisfy the specification, so if this is the specification on which the clients relied, the two implementations are equivalent and substitutable for one another.

Here is a slightly different specification:

```
static int findOneOrMore,AnyIndex(int[] arr, int val)
    requires: val occurs in arr
    effects: returns index i such that arr[i] = val
```

This specification is not deterministic. It 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 allows multiple valid outputs for the same input.

Note that this is different from *nondeterministic* in the usual sense of that word. Nondeterministic code sometimes behaves one way and sometimes another, even if called in the same program with the same inputs. This can happen, for example, when the code's behavior depends on a random number, or when

it depends on the timing of concurrent processes. But a specification which is not deterministic doesn't have to have a nondeterministic implementation. It can be satisfied by a fully deterministic implementation.

To avoid the confusion, we'll refer to specifications that are not deterministic as **underdetermined**.

This underdetermined `find` spec is again satisfied by both `find First` and `find Last`, each resolving the underdeterminedness in its own (fully deterministic) way. A client of `find OneOrMore,AnyIndex` spec can't rely on which index will be returned if `val` appears more than once. The spec would be satisfied by a nondeterministic implementation, too — for example, one that tosses a coin to decide whether to start searching from the beginning or the end of the array. But in almost all cases we'll encounter, underdeterminism in specifications offers a choice that is made by the implementor at implementation time. An underdetermined spec is typically implemented by a fully-deterministic implementation.

Declarative vs. operational specs

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 inadvertently expose implementation details 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.

One reason programmers sometimes lapse into operational specifications is because they're using the spec comment to explain the implementation for a maintainer. Don't do that. When it's necessary, use comments within the body of the method, not in the spec comment.

For a given specification, there may be many ways to express it declaratively:

```
static boolean startsWith(String str, String prefix)
  effects: returns true if and only if there exists String suffix
            such that prefix + suffix == str
```

```
static boolean startsWith(String str, String prefix)
  effects: returns true if and only if there exists integer i
            such that str.substring(0, i) == prefix
```

```
static boolean startsWith(String str, String prefix)
  effects: returns true if the first prefix.length() characters of str
            are the characters of prefix, false otherwise
```

It's up to us to choose the clearest specification for clients and maintainers of the code.

Stronger vs. weaker specs

Suppose you want to change a method – either how its implementation behaves, or the specification itself. There are already clients that depend on the method's current specification. How do you compare the behaviors of two specifications to decide whether it's safe to replace the old spec with the new spec?

A specification S2 is stronger than or equal to a specification S1 if

- S2's precondition is weaker than or equal to S1's,
and
- S2's postcondition is stronger than or equal to S1's, for the states that satisfy S1's precondition.

If this is the case, then an implementation that satisfies S2 can be used to satisfy S1 as well, and it's safe to replace S1 with S2 in your program.

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 them. And you can always strengthen the post-condition, which means making more promises.

For example, this spec for `find` :

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

can be replaced with:

```
static int findOneOrMore,AnyIndex(int[] a, int val)
    requires: val occurs at least once in a
    effects: returns index i such that a[i] = val
```

which has a weaker precondition. This in turn can be replaced with:

```
static int findOneOrMore,FirstIndex(int[] a, int val)
    requires: val occurs at least once in a
    effects: returns lowest index i such that a[i] = val
```

which has a stronger postcondition.

What about this specification:

```
static int findCanBeMissing(int[] a, int val)
    requires: nothing
    effects: returns index i such that a[i] = val,
            or -1 if no such i
```

We'll come back to `find CanBeMissing` in the exercises.

Diagramming specifications

Imagine (very abstractly) the space of all possible Java methods.

`findFirst`

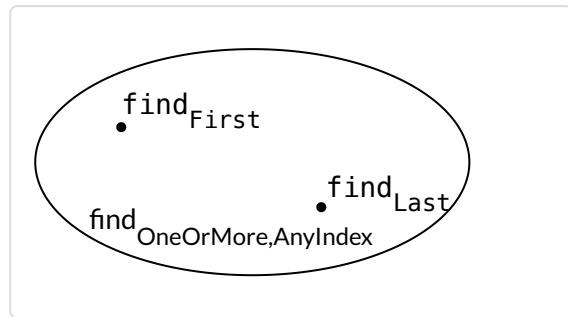
`findLast`

Each point in this space represents a method implementation.

First we'll diagram `find First` and `find Last` defined above. Look back at the code and see that `find First` and `find Last` are *not* specs. They are implementations, with method bodies that implement their actual behavior. So we denote them as points in the space.

A specification defines a *region* in the space of all possible implementations. A given implementation either behaves according to the spec, satisfying the precondition-implies-postcondition contract (it is inside the region), or it does not (outside the region).

Both `find First` and `find Last` satisfy `find OneOrMore,AnyIndex`, so they are inside the region defined by that spec.



We can imagine clients looking in on this space: the specification acts as a firewall.

- Implementors have the freedom to move around inside the spec, changing their code without fear of upsetting a client. This is crucial in order for the implementor to be able to improve the performance of their algorithm, the clarity of their code, or to change their approach when they discover a bug, etc.
- Clients don't know which implementation they will get. They must respect the spec, but also have the freedom to change how they're using the implementation without fear that it will suddenly break.

How will similar specifications relate to one another? Suppose we start with specification S1 and use it to create a new specification S2.

If S2 is stronger than S1, how will these specs appear in our diagram?

- Let's start by **strengthening the postcondition**. If S2's postcondition is now stronger than S1's postcondition, then S2 is the stronger specification.

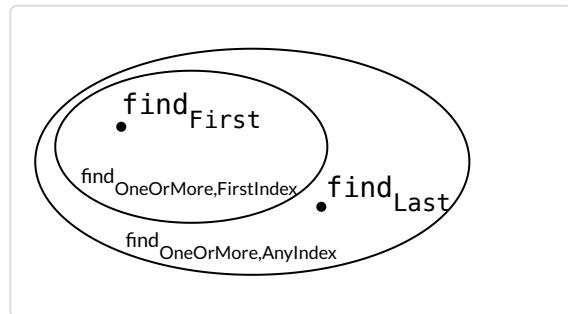
Think about what strengthening the postcondition means for implementors: it means they have less freedom, the requirements on their output are stronger. Perhaps they previously satisfied `find OneOrMore,AnyIndex` by returning any index i , but now the spec demands the *lowest* index i . So there are now implementations *inside* `find OneOrMore,AnyIndex` but *outside* `find OneOrMore,FirstIndex`.

Could there be implementations *inside* `find OneOrMore,FirstIndex` but *outside* `find OneOrMore,AnyIndex`? No. All of those implementations satisfy a stronger postcondition than what `find OneOrMore,AnyIndex` demands.

- Think through what happens if we **weaken the precondition**, which will again make S2 a stronger specification. Implementations will have to handle new inputs that were previously excluded by the spec. If they behaved badly on those inputs before, we wouldn't have noticed, but now their bad behavior is exposed.

We see that when S2 is stronger than S1, it defines a *smaller* region in this diagram; a weaker specification defines a larger region.

In our figure, since `find Last` iterates from the end of the array `arr`, it does not satisfy `find OneOrMore,FirstIndex` and is outside that region.



Another specification S3 that is neither stronger nor weaker than S1 might overlap (such that there exist implementations that satisfy only S1, only S3, and both S1 and S3) or might be disjoint. In both cases, S1 and S3 are incomparable.

Designing good specifications

What makes a good method? Designing a method means primarily writing a specification.

About the form of the specification: it should obviously be succinct, clear, and well-structured, so that it's easy to read.

The content of the specification, however, is harder to prescribe. There are no infallible rules, but there are some useful guidelines.

The specification should be coherent

The spec shouldn't have lots of different cases. Long argument lists, deeply nested if-statements, and boolean flags are all signs of trouble. Consider this specification:

```
static int sumFind(int[] a, int[] b, int val)
  effects: returns the sum of all indices in arrays a and b at which
            val appears
```

Is this a well-designed procedure? Probably not: it's incoherent, since it does several things (finding in two arrays and summing the indexes) that are not really related. It would be better to use two separate procedures, one that finds the indexes, and the other that sums them.

Here's another example, the `countLongWords` method from *Code Review* ([./04-code-review/#countLongWords](#)) :

```
public static int LONG_WORD_LENGTH = 5;
public static String longestWord;

/**
 * Update longestWord to be the longest element of words, and print
 * the number of elements with length > LONG_WORD_LENGTH to the console.
 * @param words list to search for long words
 */
public static void countLongWords(List<String> words)
```

In addition to terrible use of global variables ([./04-code-review/#dont_use_global_variables](#)) and printing instead of returning ([./04-code-review/#methods_should_return_results_not_print_them](#)), the specification is not coherent — it does two different things, counting words and finding the longest word.

Separating those two responsibilities into two different methods will make them simpler (easy to understand) and more useful in other contexts (ready for change).

The results of a call should be informative

Consider the specification of a method that puts a value in a map, where keys are of some type `K` and values are of some type `V` :

```
static V put (Map<K,V> map, K key, V val)
  requires: val may be null, and map may contain null values
  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 `null`s. 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 for sure that you didn't insert `null`s.

The specification should be strong enough

Of course the spec should give clients a strong enough guarantee in the general case — it needs to satisfy their basic requirements. We must use extra care when specifying the special cases, to make sure they don't undermine what would otherwise be a useful method.

For example, there's no point throwing an 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> list1, List<T> list2)
  effects: adds the elements of list2 to list1,
            unless it encounters a null element,
            at which point it throws a NullPointerException
```

If a `NullPointerException` is thrown, the client is left to figure out on their own which elements of `list2` actually made it to `list1`.

The specification should also 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.

The specification should use *abstract types* where possible

We saw early on in the Java Collections section of *Basic Java* (..02-basic-java/#arraylists_and_linkedslists_creating_lists) that we can distinguish between more abstract notions like a `List` or `Set` and particular implementations like `ArrayList` or `HashSet`.

Writing our specification with *abstract types* gives more freedom to both the client and the implementor. In Java, this often means using an interface type, like `Map` or `Reader`, instead of specific implementation types like `HashMap` or `FileReader`. Consider this specification:

```
static ArrayList<T> reverse(ArrayList<T> list)
  effects: returns a new list which is the reversal of list, i.e.
            newList[i] == list[n-i-1]
            for all 0 <= i < n, where n = list.size()
```

This forces the client to pass in an `ArrayList`, and forces the implementor to return an `ArrayList`, even if there might be alternative `List` implementations that they would rather use. Since the behavior of the specification doesn't depend on anything specific about `ArrayList`, it would be better to write this spec in terms of the more abstract `List`.

Precondition or postcondition?

Another design issue is whether to use a precondition, and if so, whether the method code should attempt to make sure the precondition has been met before proceeding. In fact, the most common use of preconditions is to demand a property precisely because it would be hard or expensive for the method to check it.

As mentioned above, a non-trivial precondition 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. That's why the Java API classes, for example, tend to specify (as a postcondition) that they throw unchecked exceptions when arguments are inappropriate. This approach makes it easier to find the bug or incorrect assumption in the caller code that led to passing bad arguments. In general, it's better to **fail fast**, as close as possible to the site of the bug, rather than let bad values propagate through a program far from their original cause.

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. Instead, like the Java API classes, you should throw an exception.

About access control

Read: [Packages](https://docs.oracle.com/javase/tutorial/java/package/index.html) ([//docs.oracle.com/javase/tutorial/java/package/index.html](https://docs.oracle.com/javase/tutorial/java/package/index.html)) (7 short pages) in the Java Tutorials.

Read: [Controlling Access](https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html) ([//docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html](https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html)) (1 page) in the Java Tutorials.

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

About static vs. instance methods

Read: the `static` keyword ([//www.codeguru.com/java/tij/tij0037.shtml#Heading79](http://www.codeguru.com/java/tij/tij0037.shtml#Heading79)) on CodeGuru.

We have also been using *static* for almost all of our methods, again without much discussion. Static methods are not associated with any particular instance of a class, while *instance* methods (declared without the `static` keyword) must be called on a particular object.

Specifications for instance methods are written just the same way as specifications for static methods, but they will often refer to properties of the instance on which they were called.

For example, by now we're very familiar with this specification:

```
static int find(int[] arr, int val)
  requires: val occurs in arr
  effects:  returns index i such that arr[i] = val
```

Instead of using an `int[]`, what if we had a class `IntArray` designed for storing arrays of integers? The `IntArray` class might provide an instance method with the specification:

```
int find(int val)
  requires: val occurs in this array
  effects:  returns index i such that the value at index i in this array
            is val
```

We'll have much more to say about specifications for instance methods in future classes!

Summary

A specification acts as a crucial firewall between implementor and client — both between people (or the same person at different times) and between code. As we saw last time ([./06-specifications/#summary](#)), it makes separate development possible: the client is free to write code that uses a module without seeing its source code, and the implementor is free to write the implementation code without knowing how it will be used.

Declarative specifications are the most useful in practice. Preconditions (which weaken the specification) make life harder for the client, but applied judiciously they are a vital tool in the software designer's repertoire, allowing the implementor to make necessary assumptions.

As always, our goal is to design specifications that make our software:

- **Safe from bugs** . Without specifications, even the tiniest change to any part of our program could be the tipped domino that knocks the whole thing over. Well-structured, coherent specifications minimize misunderstandings and maximize our ability to write correct code with the help of static checking, careful reasoning, testing, and code review.
- **Easy to understand** . A well-written declarative specification means the client doesn't have to read or understand the code. You've probably never read the code for, say, Python `dict.update` (<https://hg.python.org/cpython/file/7ae156f07a90/Objects/dictobject.c#l1990>) , and doing so isn't nearly as useful to the Python programmer as reading the declarative spec (<https://docs.python.org/3/library/stdtypes.html#dict.update>) .
- **Ready for change** . An appropriately weak specification gives freedom to the implementor, and an appropriately strong specification gives freedom to the client. We can even change the specs themselves, without having to revisit every place they're used, as long as we're only strengthening them: weakening preconditions and strengthening postconditions.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 8: Avoiding Debugging

First Defense: Make Bugs Impossible

Second Defense: Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental Development

Modularity & Encapsulation

Reading 8: Avoiding Debugging

6.005 Prime Objective

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

The topic of today's class is debugging – or rather, how to avoid debugging entirely, or keep it easy when you have to do it.

First Defense: Make Bugs Impossible

The best defense against bugs is to make them impossible by design.

One way that we've already talked about is **static checking** ([./01-static-checking/#static_checking_dynamic_checking_no_checking](#)) . Static checking eliminates many bugs by catching them at compile time.

We also saw some examples of **dynamic checking** in earlier class meetings. For example, Java makes array overflow bugs impossible by catching them dynamically. If you try to use an index outside the bounds of an array or a List, then Java automatically produces an error. Older languages like C and C++ silently allow the bad access, which leads to bugs and security vulnerabilities ([//en.wikipedia.org/wiki/Buffer_overflow](https://en.wikipedia.org/wiki/Buffer_overflow)) .

Immutability (immunity from change) is another design principle that prevents bugs. An *immutable* type is a type whose values can never change once they have been created.

String is an immutable type. There are no methods that you can call on a String that will change the sequence of characters that it represents. Strings can be passed around and shared without fear that they will be modified by other code.

Java also gives us *immutable references* : variables declared with the keyword `final` , which can be assigned once but never reassigned. 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.

Consider this example:

```
final char[] vowels = new char[] { 'a', 'e', 'i', 'o', 'u' };
```

The `vowels` variable is declared final, but is it really unchanging? Which of the following statements will be illegal (caught statically by the compiler), and which will be allowed?

```
vowels = new char[] { 'x', 'y', 'z' };
vowels[0] = 'z';
```

You'll find the answers in the exercise below. Be careful about what `final` means! It only makes the *reference* immutable, not necessarily the *object* that the reference points to.

Second Defense: Localize Bugs

If we can't prevent bugs, we can try to localize them to a small part of the program, so that we don't have to look too hard to find the cause of a bug. When localized to a single method or small module, bugs may be found simply by studying the program text.

We already talked about **fail fast** : the earlier a problem is observed (the closer to its cause), the easier it is to fix.

Let's begin with a simple example:

```
/**
 * @param x requires x >= 0
 * @return approximation to square root of x
 */
public double sqrt(double x) { ... }
```

Now suppose somebody calls `sqrt` with a negative argument. What's the best behavior for `sqrt` ? Since the caller has failed to satisfy the requirement that `x` should be nonnegative, `sqrt` is no longer bound by the terms of its contract, so it is technically free to do whatever it wants: return an arbitrary value, or enter an infinite loop, or melt down the CPU. Since the bad call indicates a bug in the caller, however, the most useful behavior would point out the bug as early as possible. We do this by inserting a runtime assertion that tests the precondition. Here is one way we might write the assertion:

```
/**
 * @param x requires x >= 0
 * @return approximation to square root of x
 */
public double sqrt(double x) {
    if (! (x >= 0)) throw new AssertionError();
    ...
}
```

When the precondition is not satisfied, this code terminates the program by throwing an `AssertionError` exception. The effects of the caller's bug are prevented from propagating.

Checking preconditions is an example of **defensive programming**. Real programs are rarely bug-free. Defensive programming offers a way to mitigate the effects of bugs even if you don't know where they are.

Assertions

It is common practice to define a procedure for these kinds of defensive checks, usually called `assert`:

```
assert (x >= 0);
```

This approach abstracts away from what exactly happens when the assertion fails. The failed assert might exit; it might record an event in a log file; it might email a report to a maintainer.

Assertions have the added benefit of documenting an assumption about the state of the program at that point. To somebody reading your code, `assert (x >= 0)` says "at this point, it should always be true that $x \geq 0$." Unlike a comment, however, an assertion is executable code that enforces the assumption at runtime.

In Java, runtime assertions are a built-in feature of the language. The simplest form of the `assert` statement takes a boolean expression, exactly as shown above, and throws `AssertionError` if the boolean expression evaluates to false:

```
assert x >= 0;
```

An assert statement may also include a description expression, which is usually a string, but may also be a primitive type or a reference to an object. The description is printed in an error message when the assertion fails, so it can be used to provide additional details to the programmer about the cause of the failure. The description follows the asserted expression, separated by a colon. For example:

```
assert (x >= 0) : "x is " + x;
```

If $x == -1$, then this assertion fails with the error message

```
x is -1
```

along with a stack trace that tells you where the assert statement was found in your code and the sequence of calls that brought the program to that point. This information is often enough to get started in finding the bug.

A serious problem with Java assertions is that assertions are off by default .

If you just run your program as usual, none of your assertions will be checked! Java's designers did this because checking assertions can sometimes be costly to performance. For most applications, however, assertions are *not* expensive compared to the rest of the code, and the benefit they provide in bug-checking is worth that small cost in performance.

So you have to enable assertions explicitly by passing `-ea` (which stands for *enable assertions*) to the Java virtual machine. In Eclipse, you enable assertions by going to Run → Run Configurations → Arguments, and putting `-ea` in the VM arguments box. It's best, in fact, to enable them by default by going to Preferences → Java → Installed JREs → Edit → Default VM Arguments, as you hopefully did in the Getting Started (..../getting-started/#config-eclipse) instructions.

It's always a good idea to have assertions turned on when you're running JUnit tests. You can ensure that assertions are enabled using the following test case:

```
@Test(expected=AssertionError.class)
public void testAssertionsEnabled() {
    assert false;
}
```

If assertions are turned on as desired, then `assert false` throws an `AssertionError`. The annotation (`expected=AssertionError.class`) on the test expects and requires this error to be thrown, so the test passes. If assertions are turned off, however, then the body of the test will do nothing, failing to throw the expected exception, and JUnit will mark the test as failing.

Note that the Java `assert` statement is a different mechanism from the JUnit methods `assertTrue()`, `assertEquals()`, etc. They all assert a predicate about your code, but are designed for use in different contexts. The `assert` statement should be used in implementation code, for defensive checks inside the implementation. JUnit `assert...()` methods should be used in JUnit tests, to check the result of a test. The `assert` statements don't run without `-ea`, but the JUnit `assert...()` methods always run.

What to Assert

Here are some things you should assert:

Method argument requirements, like we saw for `sqrt`.

Method return value requirements. This kind of assertion is sometimes called a *self check*. For example, the `sqrt` method might square its result to check whether it is reasonably close to `x`:

```
public double sqrt(double x) {
    assert x >= 0;
    double r;
    ... // compute result r
    assert Math.abs(r*r - x) < .0001;
    return r;
}
```

Covering all cases. If a conditional statement or switch does not cover all the possible cases, it is good practice to use an assertion to block the illegal cases:

```
switch (vowel) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u': return "A";
    default: assert false;
}
```

The assertion in the `default` clause has the effect of asserting that `vowel` must be one of the five vowel letters.

When should you write runtime assertions? As you write the code, not after the fact. When you're writing the code, you have the invariants in mind. If you postpone writing assertions, you're less likely to do it, and you're liable to omit some important invariants.

What Not to Assert

Runtime assertions are not free. They can clutter the code, so they must be used judiciously. Avoid trivial assertions, just as you would avoid uninformative comments. For example:

```
// don't do this:  
x = y + 1;  
assert x == y+1;
```

This assertion doesn't find bugs in your code. It finds bugs in the compiler or Java virtual machine, which are components that you should trust until you have good reason to doubt them. If an assertion is obvious from its local context, leave it out.

Never use assertions to test conditions that are external to your program, such as the existence of files, the availability of the network, or the correctness of input typed by a human user. Assertions test the internal state of your program to ensure that it is within the bounds of its specification. When an assertion fails, it indicates that the program has run off the rails in some sense, into a state in which it was not designed to function properly. Assertion failures therefore indicate bugs. External failures are not bugs, and there is no change you can make to your program in advance that will prevent them from happening. External failures should be handled using exceptions instead.

Many assertion mechanisms are designed so that assertions are executed only during testing and debugging, and turned off when the program is released to users. Java's assert statement behaves this way. The advantage of this approach is that you can write very expensive assertions that would otherwise seriously degrade the performance of your program. For example, a procedure that searches an array using binary search has a requirement that the array be sorted. Asserting this requirement requires scanning through the entire array, however, turning an operation that should run in logarithmic time into one that takes linear time. You should be willing (eager!) to pay this cost during testing, since it makes debugging much easier, but not after the program is released to users.

However, disabling assertions in release has a serious disadvantage. With assertions disabled, a program has far less error checking when it needs it most. Novice programmers are usually much more concerned about the performance impact of assertions than they should be. Most assertions are cheap, so they should not be disabled in the official release.

Since assertions may be disabled, the correctness of your program should never depend on whether or not the assertion expressions are executed. In particular, asserted expressions should not have *side-effects*. For example, if you want to assert that an element removed from a list was actually found in the list, don't write it like this:

```
// don't do this:  
assert list.remove(x);
```

If assertions are disabled, the entire expression is skipped, and `x` is never removed from the list. Write it like this instead:

```
boolean found = list.remove(x);  
assert found;
```

For 6.005, you are required to have assertions turned on, all the time. Make sure you did this in Eclipse, following the instructions in the [Getting Started handout](#) ([..../getting-started/#config-eclipse](#)) . If you don't have assertions turned on, you will be sad, and the staff won't have much sympathy.

Incremental Development

A great way to localize bugs to a tiny part of the program is incremental development. Build only a bit of your program at a time, and test that bit thoroughly before you move on. That way, when you discover a bug, it's more likely to be in the part that you just wrote, rather than anywhere in a huge pile of code.

Our class on testing talked about two techniques that help with this:

- **Unit testing** (../03-testing/#unit_testing_and_stubs) : when you test a module in isolation, you can be confident that any bug you find is in that unit – or maybe in the test cases themselves.
- **Regression testing** (../03-testing/#automated_testing_and_regression_testing) : when you're adding a new feature to a big system, run the regression test suite as often as possible. If a test fails, the bug is probably in the code you just changed.

Modularity & Encapsulation

You can also localize bugs by better software design.

Modularity. Modularity means dividing up a system into components, or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system. The opposite of a modular system is a monolithic system – big and with all of its pieces tangled up and dependent on each other.

A program consisting of a single, very long main() function is monolithic – harder to understand, and harder to isolate bugs in. By contrast, a program broken up into small functions and classes is more modular.

Encapsulation. Encapsulation means building walls around a module (a hard shell or capsule) so that the module is responsible for its own internal behavior, and bugs in other parts of the system can't damage its integrity.

One kind of encapsulation is access control

(//docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html), using public and private to control the visibility and accessibility of your variables and methods. A public variable or method can be accessed by any code (assuming the class containing that variable or method is also public). A private variable or method can only be accessed by code in the same class. Keeping things private as much as possible, especially for variables, provides encapsulation, since it limits the code that could inadvertently cause bugs.

Another kind of encapsulation comes from **variable scope**. The scope of a variable is the portion of the program text over which that variable is defined, in the sense that expressions and statements can refer to the variable. A method parameter's scope is the body of the method. A local variable's scope extends from its declaration to the next closing curly brace. Keeping variable scopes as small as possible makes it much easier to reason about where a bug might be in the program. For example, suppose you have a loop like this:

```
for (i = 0; i < 100; ++i) {
    ...
    doSomeThings();
    ...
}
```

...and you've discovered that this loop keeps running forever – i never reaches 100. Somewhere, somebody is changing i. But where? If i is declared as a global variable like this:

```
public static int i;
...
for (i = 0; i < 100; ++i) {
    ...
    doSomeThings();
    ...
}
```

...then its scope is the entire program. It might be changed anywhere in your program: by `doSomeThings()`, by some other method that `doSomeThings()` calls, by a concurrent thread running some completely different code. But if `i` is instead declared as a local variable with a narrow scope, like this:

```
for (int i = 0; i < 100; ++i) {
    ...
    doSomeThings();
    ...
}
```

...then the only place where `i` can be changed is within the for statement – in fact, only in the ... parts that we've omitted. You don't even have to consider `doSomeThings()`, because `doSomeThings()` doesn't have access to this local variable.

Minimizing the scope of variables is a powerful practice for bug localization. Here are a few rules that are good for Java:

- **Always declare a loop variable in the for-loop initializer.** So rather than declaring it before the loop:

```
int i;
for (i = 0; i < 100; ++i) {
```

which makes the scope of the variable the entire rest of the outer curly-brace block containing this code, you should do this:

```
for (int i = 0; i < 100; ++i) {
```

which makes the scope of `i` limited just to the for loop.

- **Declare a variable only when you first need it, and in the innermost curly-brace block that you can.** Variable scopes in Java are curly-brace blocks, so put your variable declaration in the innermost one that contains all the expressions that need to use the variable. Don't declare all your variables at the start of the function – it makes their scopes unnecessarily large. But note that in languages without static type declarations, like Python and Javascript, the scope of a variable is normally the entire function anyway, so you can't restrict the scope of a variable with curly braces, alas.
- **Avoid global variables.** Very bad idea, especially as programs get large. Global variables are often used as a shortcut to provide a parameter to several parts of your program. It's better to just pass the parameter into the code that needs it, rather than putting it in global space where it can inadvertently reassigned.

Summary

In this reading, we looked at some ways to minimize the cost of debugging:

- Avoid debugging
 - make bugs impossible with techniques like static typing, automatic dynamic checking, and immutable types and references
- Keep bugs confined
 - failing fast with assertions keeps a bug's effects from spreading
 - incremental development and unit testing confine bugs to your recent code
 - scope minimization reduces the amount of the program you have to search

Thinking about our three main measures of code quality:

- **Safe from bugs.** We're trying to prevent them and get rid of them.
- **Easy to understand.** Techniques like static typing, final declarations, and assertions are additional documentation of the assumptions in your code. Variable scope minimization makes it easier for a reader to understand how the variable is used, because there's less code to look at.
- **Ready for change.** Assertions and static typing document the assumptions in an automatically-checkable way, so that when a future programmer changes the code, accidental violations of those assumptions are detected.

6.005 – Software Construction on MIT OpenCourseWare (<https://ocw.mit.edu/ans7870/6/6.005/s16/>) |
 OCW 6.005 Homepage (<https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-software-construction-spring-2016/>) Spring 2016

Reading 9: Mutability & Immutability

Mutability

Risks of mutation

Aliasing is what makes mutable types risky

Specifications for mutating methods

Iterating over arrays and lists

Mutation undermines an iterator

Mutation and contracts

Useful immutable types

Summary

Reading 9: Mutability & Immutability

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

- Understand mutability and mutable objects
- Identify aliasing and understand the dangers of mutability
- Use immutability to improve correctness, clarity, & changeability

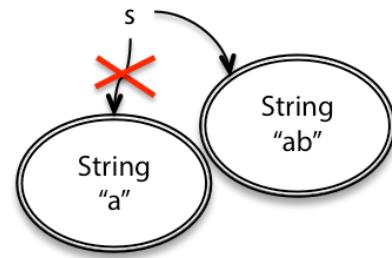
Mutability

Recall from *Basic Java* when we discussed snapshot diagrams ([./02-basic-java/#snapshot_diagrams](#)) that some objects are *immutable* : once created, they always represent the same value. Other objects are *mutable* : they have methods that change the value of the object.

`String` (<https://docs.oracle.com/javase/8/docs/api/?java/lang/String.html>) is an example of an immutable type. A `String` object always represents the same string. `StringBuilder` (<https://docs.oracle.com/javase/8/docs/api/?java/lang/StringBuilder.html>) is an example of a mutable type. It has methods to delete parts of the string, insert or replace characters, etc.

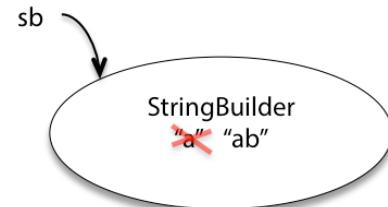
Since `String` is immutable, once created, a `String` object always has the same value. To add something to the end of a `String`, you have to create a new `String` object:

```
String s = "a";
s = s.concat("b"); // s+="b" and s=s+"b" also mean the same thing
```



By contrast, `StringBuilder` objects are mutable. This class has methods that change the value of the object, rather than just returning new values:

```
StringBuilder sb = new StringBuilder("a");
sb.append("b");
```

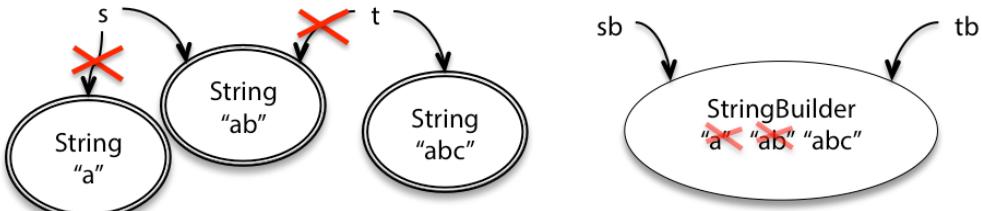


`StringBuilder` has other methods as well, for deleting parts of the string, inserting in the middle, or changing individual characters.

So what? In both cases, you end up with `s` and `sb` referring to the string of characters "ab". 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. For example, when another variable `t` points to the same `String` object as `s`, and another variable `tb` points to the same `StringBuilder` as `sb`, then the differences between the immutable and mutable objects become more evident:

```
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. Consider this code:

```
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` with a `toString()` call:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < n; ++i) {
    sb.append(String.valueOf(i));
}
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.

Risks of mutation

Mutable types seem much more powerful than immutable types. If you were shopping in the Datatype Supermarket, and you had to choose between a boring immutable `String` and a super-powerful-do-anything mutable `StringBuilder`, why on earth would you choose the immutable one?

`StringBuilder` should be able to do everything that `String` can do, plus `set()` and `append()` and everything else.

The answer is that **immutable types are safer from bugs, easier to understand, and more ready for change**. Mutability makes it harder to understand what your program is doing, and much harder to enforce contracts. Here are two examples that illustrate why.

Risky example #1: passing mutable values

Let's start with a simple method that sums the integers in a list:

```
/** @return the sum of the numbers in the list */
public static int sum(List<Integer> list) {
    int sum = 0;
    for (int x : list)
        sum += x;
    return sum;
}
```

Suppose we also need a method that sums the absolute values. Following good DRY practice (Don't Repeat Yourself ([//en.wikipedia.org/wiki/Don't_repeat_yourself](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself))), the implementer writes a method that uses `sum()`:

```
/** @return the sum of the absolute values of the numbers in the list */
public static int sumAbsolute(List<Integer> list) {
    // let's reuse sum(), because DRY, so first we take absolute values
    for (int i = 0; i < list.size(); ++i)
        list.set(i, Math.abs(list.get(i)));
    return sum(list);
}
```

Notice that this method does its job by **mutating the list directly**. It seemed sensible to the implementer, because it's more efficient to reuse the existing list. If the list is millions of items long, then you're saving the time and memory of generating a new million-item list of absolute values. So the implementer has two very good reasons for this design: DRY, and performance.

But the resulting behavior will be very surprising to anybody who uses it! For example:

```
// meanwhile, somewhere else in the code...
public static void main(String[] args) {
    // ...
    List<Integer> myData = Arrays.asList(-5, -3, -2);
    System.out.println(sumAbsolute(myData));
    System.out.println(sum(myData));
}
```

What will this code print? Will it be 10 followed by -10? Or something else?

Let's think about the key points here:

- **Safe from bugs?** In this example, it's easy to blame the implementer of `sumAbsolute()` for going beyond what its spec allowed. But really, **passing mutable objects around is a latent bug**. It's just waiting for some programmer to inadvertently mutate that list, often with very good intentions like reuse or performance, but resulting in a bug that may be very hard to track down.
- **Easy to understand?** When reading `main()`, what would you assume about `sum()` and `sumAbsolute()`? Is it clearly visible to the reader that `myData` gets *changed* by one of them?

Risky example #2: returning mutable values

We just saw an example where passing a mutable object to a function caused problems. What about returning a mutable object?

Let's consider `Date` (<https://docs.oracle.com/javase/8/docs/api/?java/util/Date.html>), one of the built-in Java classes. `Date` happens to be a mutable type. Suppose we write a method that determines the first day of spring:

```
/** @return the first day of spring this year */
public static Date startOfSpring() {
    return askGroundhog();
}
```

Here we're using the well-known Groundhog algorithm for calculating when spring starts (Harold Ramis, Bill Murray, et al. *Groundhog Day*, 1993).

Clients start using this method, for example to plan their big parties:

```
// somewhere else in the code...
public static void partyPlanning() {
    Date partyDate = startOfSpring();
    // ...
}
```

All the code works and people are happy. Now, independently, two things happen. First, the implementer of `startOfSpring()` realizes that the groundhog is starting to get annoyed from being constantly asked when spring will start. So the code is rewritten to ask the groundhog at most once, and then cache the groundhog's answer for future calls:

```
/** @return the first day of spring this year */
public static Date startOfSpring() {
    if (groundhogAnswer == null) groundhogAnswer = askGroundhog();
    return groundhogAnswer;
}
private static Date groundhogAnswer = null;
```

(Aside: note the use of a private static variable for the cached answer. Would you consider this a global variable, or not?)

Second, one of the clients of `startOfSpring()` decides that the actual first day of spring is too cold for the party, so the party will be exactly a month later instead:

```
// somewhere else in the code...
public static void partyPlanning() {
    // let's have a party one month after spring starts!
    Date partyDate = startOfSpring();
    partyDate.setMonth(partyDate.getMonth() + 1);
    // ... uh-oh. what just happened?
}
```

(Aside: this code also has a latent bug in the way it adds a month. Why? What does it implicitly assume about when spring starts?)

What happens when these two decisions interact? Even worse, think about who will first discover this bug — will it be `startOfSpring()`? Will it be `partyPlanning()`? Or will it be some completely innocent third piece of code that also calls `startOfSpring()`?

Key points:

- **Safe from bugs?** Again we had a latent bug that reared its ugly head.
- **Ready for change?** Obviously the mutation of the date object is a change, but that's not the kind of change we're talking about when we say "ready for change." Instead, the question is whether the code of the program can be easily changed without rewriting a lot of it or introducing bugs. Here we had two apparently independent changes, by different programmers, that interacted to produce a bad bug.

In both of these examples — the `List<Integer>` and the `Date` — the problems would have been completely avoided if the list and the date had been immutable types. The bugs would have been impossible by design.

In fact, you should never use `Date`! Use one of the classes from package `java.time`
`(//docs.oracle.com/javase/8/docs/api/index.html?java/time/package-summary.html)`: `LocalDateTime`
`(//docs.oracle.com/javase/8/docs/api/?java/time/LocalDateTime.html)`, `Instant`
`(//docs.oracle.com/javase/8/docs/api/?java/time/Instant.html)`, etc. All guarantee in their specifications that they are *immutable*.

This example also illustrates why using mutable objects can actually be *bad* for performance. The simplest solution to this bug, which avoids changing any of the specifications or method signatures, is for `startOfSpring()` to always return a *copy* of the groundhog's answer:

```
return new Date(groundhogAnswer.getTime());
```

This pattern is **defensive copying**, and we'll see much more of it when we talk about abstract data types. The defensive copy means `partyPlanning()` can freely stomp all over the returned date without affecting `startOfSpring()`'s cached date. But defensive copying forces `startOfSpring()` to do extra work and use extra space for every *client* — even if 99% of the clients never mutate the date it returns. We may end up with lots of copies of the first day of spring throughout memory. If we used an immutable type instead, then different parts of the program could safely share the same values in memory, so less copying and less memory space is required. Immutability can be more efficient than mutability, because immutable types never need to be defensively copied.

Aliasing is what makes mutable types risky

Actually, using mutable objects is just fine if you are using them entirely locally within a method, and with only one reference to the object. What led to the problem in the two examples we just looked at was having multiple references, also called **aliases**, for the same mutable object.

Walking through the examples with a snapshot diagram will make this clear, but here's the outline:

- In the `List` example, the same list is pointed to by both `list` (in `sum` and `sumAbsolute`) and `myData` (in `main`). One programmer (`sumAbsolute`'s) thinks it's ok to modify the list; another programmer (`main`'s) wants the list to stay the same. Because of the aliases, `main`'s programmer loses.
- In the `Date` example, there are two variable names that point to the `Date` object, `groundhogAnswer` and `partyDate`. These aliases are in completely different parts of the code, under the control of different programmers who may have no idea what the other is doing.

Draw snapshot diagrams on paper first, but your real goal should be to develop the snapshot diagram in your head, so you can visualize what's happening in the code.

Specifications for mutating methods

At this point it should be clear that when a method performs mutation, it is crucial to include that mutation in the method's spec, using the structure we discussed in the previous reading (./06-specifications/specs/#specifications_for_mutating_methods).

(Now we've seen that even when a particular method *doesn't* mutate an object, that object's mutability can still be a source of bugs.)

Here's an example of a mutating method:

```
static void sort(List<String> lst)
    requires: nothing
    effects: puts lst in sorted order, i.e. lst[i] <= lst[j]
             for all 0 <= i < j < lst.size()
```

And an example of a method that does not mutate its argument:

```
static List<String> toLowerCase(List<String> lst)
    requires: nothing
    effects: returns a new list t where t[i] = lst[i].toLowerCase()
```

If the `effects` do not explicitly say that an input can be mutated, then in 6.005 we assume mutation of the input is implicitly disallowed. Virtually all programmers would assume the same thing. Surprise mutations lead to terrible bugs.

Iterating over arrays and lists

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 (./02-basic-java/#iteration) to step through a `List` or array. This code:

```
List<String> lst = ...;
for (String str : lst) {
    System.out.println(str);
}
```

is rewritten by the compiler into something like this:

```
List<String> lst = ...;
Iterator iter = lst.iterator();
while (iter.hasNext()) {
    String str = iter.next();
    System.out.println(str);
}
```

An iterator has two methods:

- `next()` returns the next element in the collection
- `hasNext()` tests whether the iterator has reached the end of the collection.

Note that the `next()` method is a **mutator** method, not only returning an element but also advancing the iterator so that the subsequent call to `next()` will return a different element.

You can also look at the Java API definition of `Iterator` ([//docs.oracle.com/javase/8/docs/api/java/util/Iterator.html](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html)) .

Before we go any further:

You should already have read: **Classes and Objects** ([//docs.oracle.com/javase/tutorial/java/javaOO/index.html](https://docs.oracle.com/javase/tutorial/java/javaOO/index.html)) in the Java Tutorials.

Read: **the `final` keyword** ([//www.codeguru.com/java/tij/tij0071.shtml](https://www.codeguru.com/java/tij/tij0071.shtml)) on CodeGuru.

MyIterator

To better understand how an iterator works, here's a simple implementation of an iterator for `ArrayList<String>` :

`MyIterator` makes use of a few Java language features that are different from the classes we've been writing up to this point. Make sure you've read the linked Java Tutorial sections so that you understand them:

Instance variables ([//docs.oracle.com/javase/tutorial/java/javaOO/variables.html](https://docs.oracle.com/javase/tutorial/java/javaOO/variables.html)) , 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** ([//docs.oracle.com/javase/tutorial/java/javaOO/constructors.html](https://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html)) , which makes a new object instance and initializes its instance variables. Where is the constructor of `MyIterator` ?

The `static` keyword is missing from `MyIterator` 's methods, which means they are **instance methods** ([//docs.oracle.com/javase/tutorial/java/javaOO/methods.html](https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html)) that must be called on an instance of the object, e.g. `iter.next()` .

The **this keyword** ([//docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html](https://docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html)) is used at one point to refer to the **instance object** , in particular to refer to an instance variable (`this.list`). This was done to disambiguate two different variables named `list` (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., `index` actually means `this.index` .

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 (access control ([//docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html](https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html))).

final is used to indicate which of the object's internal variables can be reassigned and which can't. `index` is allowed to change (`next()` updates it as it steps through the list), but `list` 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).

```
/***
 * 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> list;
    private int index;
    // list[index] is the next element that will be returned
    // by next()
    // index == list.size() means no more elements to return

    /**
     * Make an iterator.
     * @param list list to iterate over
     */
    public MyIterator(ArrayList<String> list) {
        this.list = list;
        this.index = 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 index < list.size();
    }

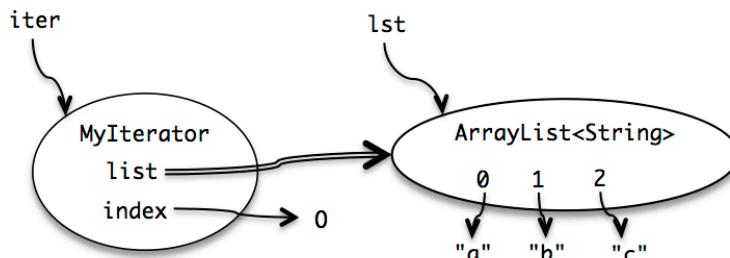
    /**
     * Get the next element of the list.
     * Requires: hasNext() returns true.
     * Modifies: this iterator to advance it to the element
     *           following the returned element.
     * @return next element of the list
     */
    public String next() {
        final String element = list.get(index);
        ++index;
        return element;
    }
}
```

Here's a snapshot diagram showing a typical state for a `MyIterator` object in action:

Note that we draw the arrow from `list` with a double line, to indicate that it's *final*. That means 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 `list` 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 that iterates over it. 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.



Mutation undermines an iterator

Let's try using our iterator for a simple job. Suppose we have a list of MIT subjects represented as strings, like `["6.005", "8.03", "9.00"]`. We want a method `dropCourse6` that will delete the Course 6 subjects from the list, leaving the other subjects behind. Following good practices, we first write the spec:

```

/**
 * Drop all subjects that are from Course 6.
 * Modifies subjects list by removing subjects that start with "6."
 *
 * @param subjects list of MIT subject numbers
 */
public static void dropCourse6(ArrayList<String> subjects)
  
```

Note that `dropCourse6` has a frame condition (the *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:
//   subjects.size: 0, 1, n
//   contents: no 6.xx, one 6.xx, all 6.xx
//   position: 6.xx at start, 6.xx in middle, 6.xx at end

// Test cases:
//   [] => []
//   ["8.03"] => ["8.03"]
//   ["14.03", "9.00", "21L.005"] => ["14.03", "9.00", "21L.005"]
//   ["2.001", "6.01", "18.03"] => ["2.001", "18.03"]
//   ["6.045", "6.005", "6.813"] => []
  
```

Finally, we implement it:

```
public static void dropCourse6(ArrayList<String> subjects) {
    MyIterator iter = new MyIterator(subjects);
    while (iter.hasNext()) {
        String subject = iter.next();
        if (subject.startsWith("6.")) {
            subjects.remove(subject);
        }
    }
}
```

Now we run our test cases, and they work! ... almost. The last test case fails:

```
// dropCourse6(["6.045", "6.005", "6.813"])
// expected [], actual ["6.005"]
```

We got the wrong answer: `dropCourse6` left a course 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 subject : subjects) {
    if (subject.startsWith("6.")) {
        subjects.remove(subject);
    }
}
```

then you'll get a `ConcurrentModificationException` ([//docs.oracle.com/javase/8/docs/api/java/util/ConcurrentModificationException.html](https://docs.oracle.com/javase/8/docs/api/java/util/ConcurrentModificationException.html)). The built-in 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 = subjects.iterator();
while (iter.hasNext()) {
    String subject = iter.next();
    if (subject.startsWith("6.")) {
        iter.remove();
    }
}
```

This is actually more efficient as well, it turns out, because `iter.remove()` already knows where the element it should remove is, while `subjects.remove()` had to search for it again.

But this doesn't fix the whole problem. What if there are other `Iterator`'s currently active over the same list? They won't all be informed!

Mutation and contracts

Mutable objects can make simple contracts very complex

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 the crucial requirement on the client that we've just discovered — that you can't modify a collection while you're iterating over it. Who takes responsibility for it? Iterator

```
//docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html) ? List  
//docs.oracle.com/javase/8/docs/api/?java/util/List.html) ? Collection  
//docs.oracle.com/javase/8/docs/api/?java/util/Collection.html) ? Can you find it?
```

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 a big cost in bug safety for doing so.

Mutable objects reduce changeability

Mutable objects make the contracts between clients and implementers more complicated, and reduce the freedom of the client and implementer to change. In other words, using *objects* that are allowed to change makes the *code* harder to change. Here's an example to illustrate the point.

The crux of our example will be the specification for this method, which looks up a username in MIT's database and returns the user's 9-digit identifier:

```
/**  
 * @param username username of person to look up  
 * @return the 9-digit MIT identifier for username.  
 * @throws NoSuchUserException if nobody with username is in MIT's database  
 */  
public static char[] getMitId(String username) throws NoSuchUserException {  
    // ... look up username in MIT's database and return the 9-digit ID  
}
```

A reasonable specification. Now suppose we have a client using this method to print out a user's identifier:

```
char[] id = getMitId("bitdiddle");  
System.out.println(id);
```

Now both the client and the implementor separately decide to make a change. The client is worried about the user's privacy, and decides to obscure the first 5 digits of the id:

```
char[] id = getMidId("bitdiddle");
for (int i = 0; i < 5; ++i) {
    id[i] = '*';
}
System.out.println(id);
```

The implementer is worried about the speed and load on the database, so the implementer introduces a cache that remembers usernames that have been looked up:

```
private static Map<String, char[]> cache = new HashMap<String, char[]>();

public static char[] getMidId(String username) throws NoSuchUserException {
    // see if it's in the cache already
    if (cache.containsKey(username)) {
        return cache.get(username);
    }

    // ... look up username in MIT's database ...

    // store it in the cache for future lookups
    cache.put(username, id);
    return id;
}
```

These two changes have created a subtle bug. When the client looks up "bitdiddle" and gets back a char array, now both the client and the implementer's cache are pointing to the *same* char array. The array is aliased. That means that the client's obscuring code is actually overwriting the identifier in the cache, so future calls to `getMidId("bitdiddle")` will not return the full 9-digit number, like "928432033", but instead the obscured version "*****2033".

Sharing a mutable object complicates a contract. If this contract failure went to software engineering court, it would be contentious. Who's to blame here? Was the client obliged not to modify the object it got back? Was the implementer obliged not to hold on to the object that it returned?

Here's one way we could have clarified the spec:

```
public static char[] getMidId(String username) throws NoSuchUserException
    requires: nothing
    effects: returns an array containing the 9-digit MIT identifier of username,
            or throws NoSuchUserException if nobody with username is in MIT's
            database. Caller may never modify the returned array.
```

This is a bad way to do it. The problem with this approach is that it means the contract has to be in force for the entire rest of the program. It's a lifetime contract! The other contracts we wrote were much narrower in scope; you could think about the precondition just before the call was made, and the postcondition just after, and you didn't have to reason about what would happen for the rest of time.

Here's a spec with a similar problem:

```
public static char[] getMidId(String username) throws NoSuchUserException
    requires: nothing
    effects: returns a new array containing the 9-digit MIT identifier of username,
            or throws NoSuchUserException if nobody with username is in MIT's
            database.
```

This doesn't entirely fix the problem either. This spec at least says that the array has to be fresh. But does it keep the implementer from holding an alias to that new array? Does it keep the implementer from changing that array or reusing it in the future for something else?

Here's a much better spec:

```
public static String getMitId(String username) throws NoSuchUserException
    requires: nothing
    effects: returns the 9-digit MIT identifier of username, or throws
             NoSuchUserException if nobody with username is in MIT's database.
```

The immutable String return value provides a *guarantee* that the client and the implementer will never step on each other the way they could with char arrays. It doesn't depend on a programmer reading the spec comment carefully. String is *immutable*. Not only that, but this approach (unlike the previous one) gives the implementer the freedom to introduce a cache — a performance improvement.

Useful immutable types

Since immutable types avoid so many pitfalls, let's enumerate some commonly-used immutable types in the Java API:

- The primitive types and primitive wrappers are all immutable. If you need to compute with large numbers, `BigInteger` ([//docs.oracle.com/javase/8/docs/api/?java/math/BigInteger.html](https://docs.oracle.com/javase/8/docs/api/?java/math/BigInteger.html)) and `BigDecimal` ([//docs.oracle.com/javase/8/docs/api/?java/math/BigDecimal.html](https://docs.oracle.com/javase/8/docs/api/?java/math/BigDecimal.html)) are immutable.
- Don't use mutable `Date`s, use the appropriate immutable type from `java.time` ([//docs.oracle.com/javase/8/docs/api/index.html?java/time/package-summary.html](https://docs.oracle.com/javase/8/docs/api/index.html?java/time/package-summary.html)) based on the granularity of timekeeping you need.
- The usual implementations of Java's collections types — `List`, `Set`, `Map` — are all mutable: `ArrayList`, `HashMap`, etc. The `Collections` ([//docs.oracle.com/javase/8/docs/api/?java/util/Collections.html](https://docs.oracle.com/javase/8/docs/api/?java/util/Collections.html)) utility class has methods for obtaining *unmodifiable* views of these mutable collections:
 - `Collections.unmodifiableList`
 - `Collections.unmodifiableSet`
 - `Collections.unmodifiableMap`

You can think of the unmodifiable view as a wrapper around the underlying list/set/map. A client who has a reference to the wrapper and tries to perform mutations — `add`, `remove`, `put`, etc. — will trigger an `UnsupportedOperationException` ([//docs.oracle.com/javase/8/docs/api/?java/lang/UnsupportedOperationException.html](https://docs.oracle.com/javase/8/docs/api/?java/lang/UnsupportedOperationException.html)).

Before we pass a mutable collection to another part of our program, we can wrap it in an unmodifiable wrapper. We should be careful at that point to forget our reference to the mutable collection, lest we accidentally mutate it. (One way to do that is to let it go out of scope.) Just as a mutable object behind a `final` reference can be mutated, the mutable collection inside an unmodifiable wrapper can still be modified by someone with a reference to it, defeating the wrapper.

- `Collections` also provides methods for obtaining immutable empty collections: `Collections.emptyList`, etc. Nothing's worse than discovering your *definitely very empty* list is suddenly *definitely not empty*!

Summary

In this reading, 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.

Make sure you understand the difference between an immutable *object* (like a `String`) and an immutable *reference* (like a `final` variable). Snapshot diagrams can help with this understanding. Objects are values, represented by circles in a snapshot diagram, and an immutable one has a double border indicating that it never changes its value. A reference is a pointer to an object, represented by an arrow in the snapshot diagram, and an immutable reference is an arrow with a double line, indicating that the arrow can't be moved to point to a different object.

The key design principle here is **immutability**: using immutable objects and immutable references as much as possible. Let's review how immutability helps with the main goals of this course:

- **Safe from bugs**. Immutable objects aren't susceptible to bugs caused by aliasing. Immutable references always point to the same object.
- **Easy to understand**. Because an immutable object or reference always means the same thing, it's simpler for a reader of the code to reason about — they don't have to trace through all the code to find all the places where the object or reference might be changed, because it can't be changed.
- **Ready for change**. If an object or reference can't be changed at runtime, then code that depends on that object or reference won't have to be revised when the program changes.