

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
2n, 2n-1, ..., 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. 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:

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

```
// 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. There's no good reason for this, and personally I find it annoying.
- 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 number.

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.

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

Static Checking, Dynamic Checking, No Checking

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

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.

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:

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

- **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.

READING EXERCISES

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

You are not
logged in.

1

```
int n = 5;
if (n) {
    n = n + 1;
}
```

✓ static error

dynamic error

no error, wrong answer

➤ This is a static type error, because the `if` statement requires an expression of `boolean` type, but `n` has `int` type.

CHECK

EXPLAIN

2

```
int big = 200000; // 200,000
big = big * big; // big should be 4 billion now
```



- static error
- dynamic error
- no error, wrong answer



This is an integer overflow, because an `int` value can't represent a number bigger than 2^{31} (about 2 billion). It isn't caught statically, but unfortunately in Java it isn't caught dynamically either. Integer overflows quietly produce the wrong answer.

[CHECK](#)
[EXPLAIN](#)

3

```
double probability = 1/5;
```



- static error
- dynamic error
- no error, wrong answer



If the programmer's intent was to get 0.2, this is using the wrong operation. `/` is overloaded for both integer division and floating point division. But because it's being called with integers, 1 and 5, Java chooses integer division, and it quietly truncates the fraction and produces the wrong answer: 0.

[CHECK](#)
[EXPLAIN](#)

4

```
int sum = 0;  
int n = 0;  
int average = sum/n;
```

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

static error

dynamic error

no error, wrong answer

Division by zero can't produce an integer, so it produces a dynamic error instead.

CHECK

EXPLAIN

5

```
double sum = 7;  
double n = 0;  
double average = sum/n;
```

static error

dynamic error

no error, wrong answer

Division by zero can't produce a real number either – but unlike real numbers, double has a special value for `POSITIVE_INFINITY`, so that's what it returns when you divide a positive integer by zero. If the code is trying to compute an average value, infinity is unlikely to be a correct or useful answer.

CHECK

EXPLAIN

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];
```

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

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()`

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

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 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; as far as I know, the only reason for this requirement is to remind the programmer that the list actually contains objects, which use more memory than primitive values. 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:

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

```
// 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 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.

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

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

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

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

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

Immutability (immunity from change) is a major design principle in this course. Immutable types are types whose values can never change once they have been created. (At least not in a way that's visible to the outside world – there are some subtleties there that we'll talk more about 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:

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

- 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.

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

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.

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

Mutatively vs. Reassigning Variables

with contributions from:
Saman Amarasinghe, Adam
Chlipala, Srinivas Devadas,
Documenting Max
Edelman, John Guttag,
Daniel Jackson, Rob Miller,
Martin Rinard, and
Hacking vs. Engineering
Armando Solar-Lezama.

This work is licensed under
The Goal of 6.005
CC BY-SA 4.0

(<http://creativecommons.org/licenses/by/>-

**Why we use Java in this
course**

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.

Reading 2: Basic Java

Getting started with the
Java Tutorials

Snapshot diagrams

Java Collections

Java API
documentation

An exercise for the
reader

Reading 2: Basic Java

Deadline #1: you must complete **PS0 Part I** (<http://web.mit.edu/6.005/www/fa15/psets/ps0/>) before this class on Friday, September 11 at 1:00 pm.

Deadline #0: you must also complete the **reading exercises** in this reading by Thursday, September 10 at 10:00 pm. The reading exercises are graded solely on completion, never on correctness, as described in the **course general info** (http://web.mit.edu/6.005/www/fa15/general/#classes_readings_and_nanoquizzes).

Getting credit: on the right is a big red **Log in** button. You will only receive credit for reading exercises if you are logged in when you do them.

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** (<http://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 (<http://web.mit.edu/6.005/www/fa15/getting-started/java.html>) as an alternative resource.

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

Language basics

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

Read **Language Basics** (<http://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
(http://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_variables.html)
- Questions: Operators
(http://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_operators.html)
- Questions: Expressions, Statements, Blocks
(http://docs.oracle.com/javase/tutorial/java/nutsandbolts/QandE/questions_expressions.html)
- Questions: Control Flow
(http://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:

READING EXERCISES

Language basics

Suppose we're editing the body of a function in Java, declaring and using local variables.

```
int a = 5;      // (1)
if (a > 10) {  // (2)
    int b = 2; // (3)
} else {        // (4)
    int b = 4; // (5)
}              // (6)
b *= 3;        // (7)
```

Which line of Java code causes a compilation error?



1

v 7

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

CHECK

EXPLAIN

Fix the bug

Select the **smallest** set of changes that will fix the bug:

- Declare int b; after line 1
- Assign b = 0; before line 2
- Assign b = 2; instead of line 3
- Assign b = 4; instead of line 5
- Declare and assign int b *= 3; instead of line 7

► We need to declare b outside the scope of the if-else. However, we do not need to assign it a value. The declarations on lines 3 and 5 must be changed to assignments, but the reassignment on line 7 is fine.

CHECK

EXPLAIN

Who are you again?

If we make the required changes above, what will happen if we comment out the else clause from the if-else?

- b will be 0
- b will be 3
- b will be 6
- We will receive an error from the Java compiler, before we run the program
- We will receive an error when we run the program, before we reach the last line
- We will receive an error when we run the program, when we reach the last line

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

```
int a = 5;
int b;
if (a > 10) {
    b = 2;
} else {
    // b = 4;
}
b *= 3;
```

In addition to requiring that variables be *declared*, the Java compiler must also be certain that every variable has been assigned a value before we attempt to access its value.

For this code, the compiler sees that `b` is assigned in only one branch of the if-else, and therefore `b` might not have been assigned any value when we reach the last line.

CHECK

EXPLAIN

Numbers and strings

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

READING EXERCISES

Numbers and strings

Does this Python code give an accurate conversion from Fahrenheit to Celsius?

```
fahrenheit = 212.0
celsius = (fahrenheit - 32) * 5/9
```

✓ Yes

- No: integer arithmetic will cause `celsius` to be zero
- No: integer arithmetic will cause `celsius` to be rounded down

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

➤ The author of this code has used whitespace to sow doubt and confusion :(

`fahrenheit` is a floating point number, so `fahrenheit - 32` gives a floating point result. Order of operations says we first multiply that number by 5 (floating point result) then divide by 9 (floating point again).

CHECK

EXPLAIN

Double shot

Rewrite the first line in Java:

- ✗
- `int fahrenheit = 212.0;`
 - `Integer fahrenheit = 212.0;`
 - `float fahrenheit = 212.0;`
 - `Float fahrenheit = 212.0;`
 - `double fahrenheit = 212.0;`
 - `Double fahrenheit = 212.0;`

And the second line, where `???` is the same type you selected above:

- ✗
- `??? celsius = (fahrenheit - 32) * 5/9;`
 - `??? celsius = (fahrenheit - 32) * (5 / 9);`
 - `??? celsius = (fahrenheit - 32) * (5. / 9);`

➤ Generally, the appropriate Java type for floating point numbers is `double`. Every primitive type has a corresponding object type. We must use the object types in certain situations, but we usually don't declare variables of those types.

In the second line, let's make the formula look both familiar and correct: option 2 is incorrect due to integer division; the option 1 might *look* incorrect, whereas option 3 is clear and unambiguous.

CHECK

EXPLAIN

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

How should we print the result?

- ✗ System.out.println(fahrenheit, " -> ", celsius);
 System.out.println(fahrenheit + " -> " + celsius);
 System.out.println("%s -> %s" % (fahrenheit, celsius));
 System.out.println(Double.toString(fahrenheit) + " -> " + Double.toString(celsius));

➤ This sort of syntax works in Python:

```
print fahrenheit, " -> ", celsius
print "%s -> %s" % (fahrenheit, celsius)
```

... but not in Java.

The last option is correct code, but unnecessarily verbose. Using the `+` operator with a string and a number is illegal in Python, but in Java, the number will be converted to a `String` automatically.

CHECK

EXPLAIN

Classes and objects

Read **Classes and Objects** (<http://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 (<http://docs.oracle.com/javase/tutorial/java/javaOO/QandE/creating-questions.html>)
- Questions: Objects (<http://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.

READING EXERCISES

Classes and objects

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

```
class Tortoise:  
    def __init__(self):  
        self.position = 0  
  
    def forward(self):  
        self.position += 1  
  
pokey = Tortoise()  
pokey.forward()  
print pokey.position
```

If we translate **Tortoise** to Java, how do we declare the class?



public class Tortoise

public class Tortoise

➤ Most likely public class Tortoise { /* implementation here */ }

CHECK

EXPLAIN

Under construction

In Python we declare an `__init__` function to initialize new objects.

What will the equivalent declaration look like in Java **Tortoise**?



Tortoise t = new Tortoise()

public Tortoise()

➤ We'll discuss what `public` means later, but `public Tortoise() { /* implementation */ }` is most likely.

And how can we obtain a reference to a new **Tortoise** object?



new Tortoise()

Tortoise t = new Tortoise()

➤ `new Tortoise()` creates the new instance. If we want a reference to the instance, we must assign it to a variable — here, `t`.

CHECK**EXPLAIN**

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

Methodical

To declare the `forward` method on `Tortoise` objects in Java:

```
public void forward() {  
    // self.position += 1 (Python)  
}
```

What's the appropriate line of code for the body of the method? (check all that apply)

- ✗ `position += 1;`
- `self.position += 1;`
- `this.position += 1;`
- `Tortoise.position += 1;`

➤ Java's `this` is like `self` in Python...

But we don't specify `this` as an argument: `this` is automatically in scope in the body of an object method.

And we aren't required to write `this.` to access a field: it's implicit. However, if a local variable of the same name is in scope, then that local variable will take precedence. To access the field instead, we will have to write `this..`

CHECK**EXPLAIN**

On your mark

In Python, we used `self.position = 0` to give `Tortoise` objects a `position` that starts at zero.

In Java, we can do this either in one line:

Which of the options initializes `position` in one line?

- ✓ 1
- 2
- 3

- 4
- 5
- 6

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

... or in a combination of lines:

Which of the options initializes position using two lines?

- ✗ 1
- 2
- 3
- 4
- 5

```
public class Tortoise {  
  
    private int position = 0;          // (1)  
    static int position = 0;          // (2)  
  
    public Tortoise() {  
        int position = 0;              // (3)  
        int self.position = 0;         // (4)  
        int this.position = 0;         // (5)  
        int Tortoise.position = 0;     // (6)  
    }  
    // ...  
}
```

```
public class Tortoise {  
  
    private int position;            // (1)  
    static int position;             // (2)  
  
    public Tortoise() {  
        self.position = 0;           // (3)  
        this.position = 0;           // (4)  
        Tortoise.position = 0;       // (5)  
    }  
    // ...  
}
```

➤ Like local variables, we can declare and initialize fields at the same time.

static would mean the field is shared by *all* Tortoise instances. private we'll discuss soon, but it prevents other code from modifying the position of our Tortoise .

And in the constructor, we could simply write position = 0; but often we write this explicitly. When you start writing constructors that take parameters, you'll quickly discover why.

CHECK

EXPLAIN

Hello, world!

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

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

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.

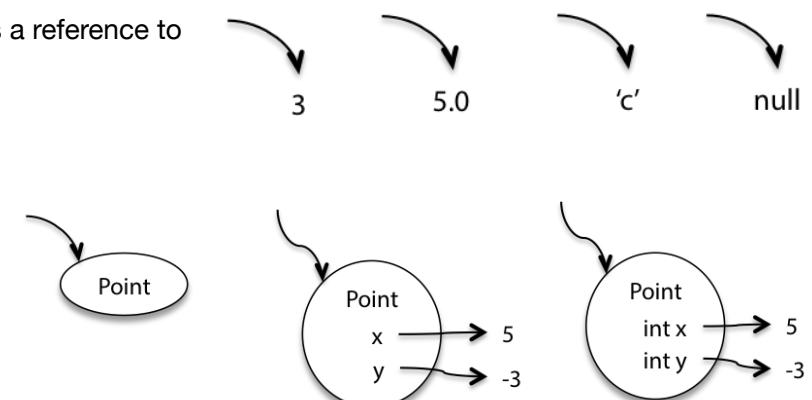
Although the diagrams in this course use examples from Java, the notation can be applied to any modern programming language, e.g., Python, Javascript, C++, Ruby.

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:

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

- 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` (<http://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` (<http://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:

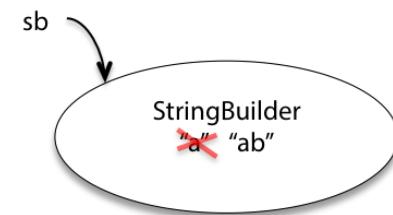
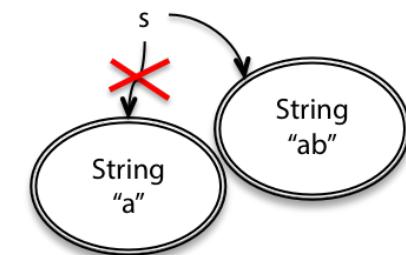
```
StringBuilder sb = new StringBuilder("a");
sb.append("b");
```

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

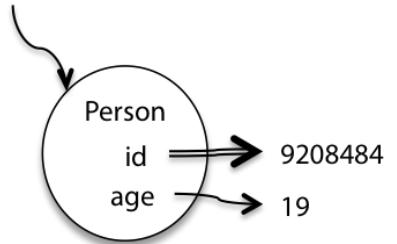
Java also gives us immutable references: variables that are assigned once and never reassigned. To make a reference immutable, declare it with the keyword `final`:

```
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**

(<http://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` (<http://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.

Reading 2: Basic Java

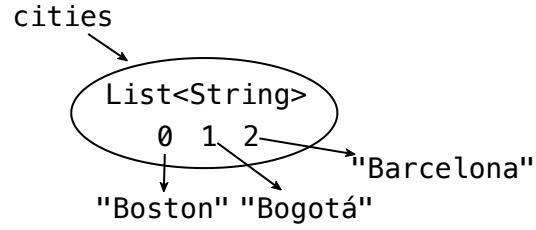
Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

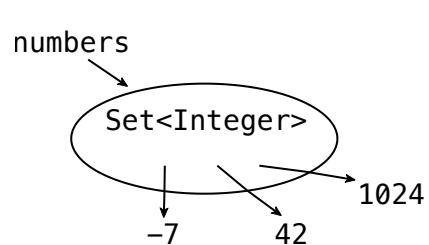


A **Set** (<http://docs.oracle.com/javase/8/docs/api/?java/util/Set.html>) is a collection of zero or more **unique objects**. Like a mathematical set or a Python set (<https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset>) – and unlike a **List** – an object cannot appear in a set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

Reading 2: Basic Java

Getting started with the Java Tutorials

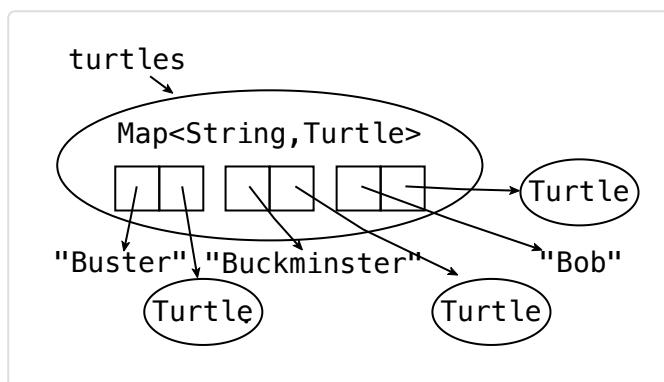
Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

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 (<http://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" })
```

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

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 types 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<>();
```

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

ArrayList (<http://docs.oracle.com/javase/8/docs/api/?java/util/ArrayList.html>) and LinkedList (<http://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 (<http://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 (<http://docs.oracle.com/javase/8/docs/api/?java/util/SortedSet.html>) with the TreeSet (<http://docs.oracle.com/javase/8/docs/api/?java/util/TreeSet.html>) implementation.

And for a Map the default choice is HashMap (<http://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])
```

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

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` (<http://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.

READING EXERCISES

Collections

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

Rewrite these variable declarations using `List` instead of arrays. We're only declaring the variables, not initializing them with any value.

`String[] names;`



`List<String> name;`

`List<String> names;`



The translation from `String[]` to `List<String>` is pretty straightforward in Java.

`int[] numbers;`



`List<Integer> numbers;`

`List<Integer> numbers;`



We can create an array of primitive types, but not a `List`. Use the `Integer` wrapper.

`char[][] grid;`



`List<List<Character>> grid;`

`List<List<Character>> grid;`



There's nothing wrong with a `List<List<Character>>` — but if this is a fixed-size grid, it might be simpler to use a 2-dimensional array instead of a list-of-lists.

CHECK

EXPLAIN

X marks the spot

Java `Map`s work like Python dictionaries.

After we run this code:

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

```
Map<String, Double> treasures = new HashMap<>();  
String x = "palm";  
treasures.put("beach", 25.);  
treasures.put("palm", 50.);  
treasures.put("cove", 75.);  
treasures.put("x", 100.);  
treasures.put("palm", treasures.get("palm") + treasures.size());  
treasures.remove("beach");  
double found = 0;  
for (double treasure : treasures.values()) {  
    found += treasure;  
}
```

What is the value of...

`treasures.get(x)`



50

54

`treasures.get("x")`



54

100

`found`



22

229

CHECK

EXPLAIN

Java API documentation

The previous section has a number of links to documentation for classes that are part of the Java platform API (<http://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** (<http://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".

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

- **java.lang.Integer** (<http://docs.oracle.com/javase/8/docs/api/?java/lang/Integer.html>) and the other primitive wrapper classes. Java automagically converts between primitive and wrapped (or “boxed”) types in most situations.
- **java.util.List** (<http://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** (<http://docs.oracle.com/javase/8/docs/api/?java/util/Map.html>), too.
- **java.io.File** (<http://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** (<http://docs.oracle.com/javase/8/docs/api/?java/io/FileReader.html>) lets us read text files.
- **java.io.BufferedReader** (<http://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` (<http://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` (<http://docs.oracle.com/javase/8/docs/api/?java/util/HashMap.html>) is an implementation of `Map` (<http://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. Constructors aren't the only way to get a new object in Java, but they are the most common.

The screenshot shows the Java API documentation for the `BufferedReader` class. At the top, there is a navigation bar with tabs for OVERVIEW, PACKAGE, CLASS (which is highlighted in orange), USE, TREE, DEPRECATED, INDEX, and HELP. Below the navigation bar, there are links for PREV CLASS and NEXT CLASS, and buttons for FRAMES, NO FRAMES, and ALL CLASSES. Summary links include NESTED | FIELD | CONSTR | METHOD, and DETAIL: FIELD | CONSTR | METHOD. The main content area starts with the package name `java.io` and the class name `Class BufferedReader`. It lists `java.lang.Object` as the superclass and `java.io.Reader` and `java.io.BufferedReader` as interfaces implemented. It also lists `Closable`, `AutoCloseable`, and `Readable` as interfaces. Below this, it lists `LineNumberReader` as a direct known subclass. The code section contains the definition of the `public class BufferedReader extends Reader`. It explains that the buffer size is specified or default, and that buffering is efficient for most purposes. It notes that each read request causes a corresponding read request to be made of the underlying character or byte stream, which is advisable to wrap a `BufferedReader` around any `Reader` whose `read()` operations may be costly, such as `FileReaders` and `InputStreamReaders`. An example code snippet shows creating a `BufferedReader` from a `FileReader`:

```
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.

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.

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

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 <code>BufferedReader</code> .	
void	<code>mark(int readheadLimit)</code> Marks the present position in the stream.	
boolean	<code>markSupported()</code> Tells whether this stream supports the <code>mark()</code> 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 **description**: read me.
- **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.

READING EXERCISES

Reading Javadocs

Use the Java API docs to answer...

Suppose we have a class `TreasureChest`. After we run this code:

You are not logged in.

```
Map<String, TreasureChest> treasures = new HashMap<>();  
treasures.put("beach", new TreasureChest(25));  
TreasureChest result = treasures.putIfAbsent("beach", new TreasureChest(75));
```

Reading 2: Basic Java

Getting started with the Java Tutorials

Snapshot diagrams

Java Collections

Java API documentation

An exercise for the reader

What is result ?

25 treasure

75 treasure

another amount of treasure

null

According to the `putIfAbsent` documentation: “If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, **else returns the current value.**”

CHECK

EXPLAIN

Avast!

After we run this code, where ??? is the appropriate type:

```
Map<String, String> translations = new HashMap<>();  
translations.put("green", "verde");  
??? result = translations.replace("green", "verde", "ahdar");
```

What is result ?

"green"

"verde"

"ahdar"

true

false

1

null

there is no appropriate type for the ???

CHECK

EXPLAIN

Reading 2: Basic Java

Getting started with the
Collaboratively authored
Java Tutorials

with contributions from:

Saman Amarasinghe, Adam
Chippala, Srikanth Devadas,

Michael Ernst, Max
Java Collections

Daniel Jackson, Rob Miller,
Martin Rinard, and

Armando Solar-Lezama.
documentation

This work is licensed under
CC BY-SA 4.0

An exercise for the
reader

<https://creativecommons.org/licenses/by-sa/4.0/>

An exercise for the reader

At this point you should have completed all the reading exercises above.

To check your reading exercise status, see classes/02-basic-java/reading-exercises on Omnivore
(<https://omni.csail.mit.edu/6.005/fa15/user/classes/02-basic-java/reading-exercises>).

Completing the reading exercises prepares you for the *nanoquiz* at the beginning of each class meeting, and
submitting the exercises is required by 10pm the evening before class.

MIT EECS accessibility (<https://accessibility.mit.edu>)

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

Reading 3: Testing

You must complete the **reading exercises** in this reading by 10:00 pm the night before class. Don't forget to log in using the big red **Log in** button. You will only receive credit for reading exercises if you are logged in when you do them.

Our Prime Objective in 6.005

Learning to write code that is:

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 (http://www.csail.mit.edu/crash_tolerant_data_storage).

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

- **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 (<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.

READING EXERCISES

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

Testing basics

In the 1990s, the Ariane 5 launch vehicle, designed and built for the European Space Agency, self-destructed 37 seconds after its first launch.

The reason was a control software bug that went undetected. The Ariane 5's guidance software was reused from the Ariane 4, which was a slower rocket. When the velocity calculation converted from a 64-bit floating point number (a `double` in Java terminology, though this software wasn't written in Java) to a 16-bit signed integer (a `short`), it overflowed the small integer and caused an exception to be thrown. The exception handler had been disabled for efficiency reasons, so the guidance software crashed. Without guidance, the rocket crashed too. The cost of the failure was \$1 billion.

What ideas does this story demonstrate?

- ✖ Even high-quality safety-critical software may still have residual bugs.
 - Testing all possible inputs is the best solution to this problem.
 - Software exhibits discontinuous behavior, unlike many physically-engineered systems.
 - Static type checking could have detected this bug.
- Testing all inputs is not feasible, because even just one 64-bit floating point number has 2^{64} possible values, which is more than the age of the universe in microseconds.
- Static type checking wouldn't have detected this bug, because the code intentionally converted a 64-bit double into a 16-bit short.

CHECK

EXPLAIN

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

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

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.

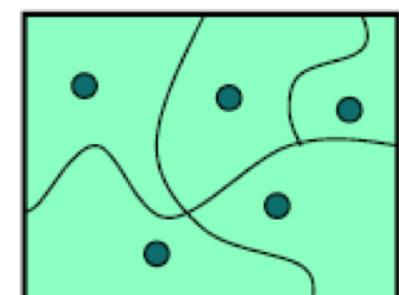
Writing tests first is a good way to understand the specification. The specification can be buggy, too — incorrect, incomplete, ambiguous, missing corner cases. Trying to write tests can uncover these problems early, before you've wasted time writing an implementation of a buggy spec.

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.



Example: `BigInteger.multiply()`

Let's look at an example. `BigInteger` (<http://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. `BigInteger` 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

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

- 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} .

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

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:

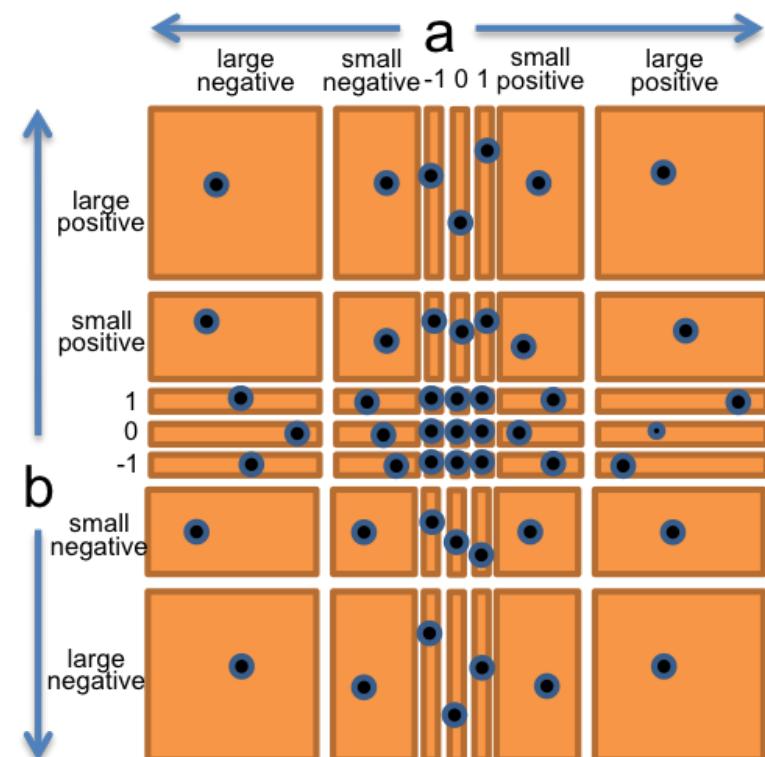
- 0
- 1
- -1
- small positive integer
- small negative integer
- huge positive integer
- huge negative integer

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 (<http://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:

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

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

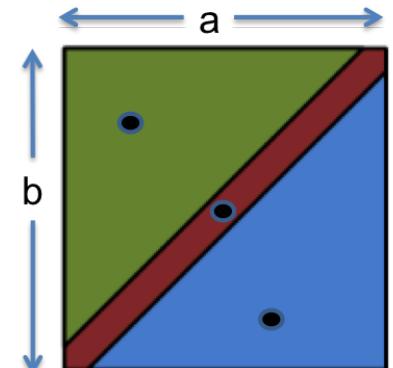
Automated Testing and Regression Testing

Summary

An exercise for the reader

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*
 - $a = 0$
 - $a < 0$

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

- a > 0
- a = minimum integer
- a = maximum integer
- value of b
 - b = 0
 - b < 0
 - b > 0
 - b = minimum integer
 - b = 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 = minint, b = maxint
- (Integer.MAX_VALUE, Integer.MIN_VALUE) covers a > b, a = maxint, b = 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.

READING EXERCISES

Partitioning

Consider the following specification:

```
/**  
 * Reverses the end of a string.  
 *  
 *          012345          012345  
 * 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 re  
 versed  
 */  
public static String reverseEnd(String text, int start)
```

Which of the following are reasonable partitions for the `start` parameter?

- start = 0, start = 5, start = 100
 - start < 0, start = 0, start > 0
 - start = 0, 0 < start < text.length(), start = text.length()
 - start < text.length(), start = text.length(), start > text.length()
- 0, 5, 100 is not a partition. A partition should be a division of the whole space of possible start values, not specific test cases.
- start < 0 and start > text.length() are not legal inputs for the function, so it isn't reasonable to use them as part of a partition. A test case must obey the requirements of the function's specification.

CHECK

EXPLAIN

Partitioning a String

Which of the following are reasonable partitions for the `text` parameter?

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

- ✖ text contains some letters; text contains no letters, but some numbers; text contains neither letters nor numbers

- `text.length() = 0; text.length > 0`

- `text.length() = 0; text.length()-start is odd; text.length()-start is even`

- text is every possible string from length 0 to 100

➤ Letters and numbers aren't important to the behavior of this function, so it isn't reasonable or necessary to partition on that property.

Length is a useful partition, however, since it can interact with the `start` parameter.

Partitioning on even and odd length is also reasonable, because reversing an odd-length substring has different behavior (since it leaves the middle element in place) than an even-length string (where all elements swap).

Partitioning into all possible length 0 to 100 strings will produce far too many test cases.

CHECK

EXPLAIN

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.

READING EXERCISES

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

Blackbox and whitebox testing

Consider the following function:

```
/**  
 * Sort a list of integers in nondecreasing order. Modifies the list so that  
 * values.get(i) <= values.get(i+1) for all 0<=i<values.length()-1  
 */  
public static void sort(List<Integer> values) {  
    // choose a good algorithm for the size of the list  
    if (values.length() < 10) {  
        radixSort(values);  
    } else if (values.length() < 1000*1000*1000) {  
        quickSort(values);  
    } else {  
        mergeSort(values);  
    }  
}
```

Which of the following test cases are likely to be boundary values produced by white box testing?

- ✖ values = [] (the empty list)
 values = [1, 2, 3]
 values = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0] ↗
 values = [0, 0, 1, 0, 0, 0, 0]
- When the list is length 10, the function switches from one sorting algorithm (radix sort) to another (quick sort). This makes length-10 lists a boundary value in the behavior of the function. But this boundary isn't visible from the function's specification (black box), only from its implementation (white box).

The empty list, on the other hand, is a boundary value whenever you're working with a list type, so white box testing isn't required to discover it.

The other lists are not boundary values by any visible evidence in this code.

CHECK

EXPLAIN

Coverage

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

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 coverage tool for Eclipse is EclEmma (<http://www.eclemma.org/>), shown below.

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

The screenshot shows the Eclipse IDE interface. At the top, there are tabs for Quoter.java, QuoteApp.java, myQuotes.rtf, FileSequenceReader.java, and Hailstone.java. Below the tabs, a code editor displays Java code for a class named FileSequenceReader. The code reads data from a sequence file using a DataInputStream. A try-catch block handles EOFException. Inside the try block, a while loop reads data into a byte array 'data' until 'read' equals 'size'. The 'justRead' variable is used to check if the read operation was successful. If it's -1, an EOFException is thrown. The code editor highlights executed lines in green and unexecuted lines in red.

Below the code editor is the Eclipse Coverage tool interface. It shows a tree view of the 'multipart' package structure under 'src'. To the right is a table with columns: Element, Coverage, Covered Instructions, and Total Instructions. The table data is as follows:

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	599

Notice how 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.

READING EXERCISES

Using a coverage tool

Install EclEmma (<http://www.eclemma.org/installation.html>) in Eclipse on your laptop. Use your laptop, because you'll need it for Testing exercises in class too.

Then create a new Java class called Hailstone.java containing this code:

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

```
public class Hailstone {  
    public static void main(String[] args) {  
        int n = 3;  
        while (n != 1) {  
            if (n % 2 == 0) {  
                n = n / 2;  
            } else {  
                n = 3 * n + 1;  
            }  
        }  
    }  
}
```

Run this class with EclEmma code coverage highlighting turned on, by choosing Run → Coverage As → Java Application.

By changing the initial value of `n`, you can observe how EclEmma highlights different lines of code differently.

When `n=3` initially, what color is the line `n = n/2` after execution?



yellow



green



Run EclEmma and find out!

If it's hard to perceive the difference between the red and green highlighting, you can change the color of the coverage highlighting by going to Preferences → General → Appearance → Editors → Text Editors → Annotations. The annotations to change are called Full Coverage, Partial Coverage, and No Coverage. You might, for example, make Full Coverage white, Partial Coverage light gray, and No Coverage dark gray.

When `n=16` initially, what color is the line `n = 3 * n + 1` after execution?



green



red



Run EclEmma and find out!

What initial value of `n` would make the line `while (n != 1)` yellow after execution?



n

1

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

- Yellow is used for a line that contains a branch (e.g., an if or while predicate) that is only taken in one direction during the program – either the predicate is false every time the program gets there, or the predicate is true every time. In this case, the way to make the while predicate yellow is to make `n != 1` false, so that the predicate executes just once and the loop is skipped entirely. `n=1` does that.

CHECK

EXPLAIN

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:

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

```
/** @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** (http://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

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

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.

READING EXERCISES

Regression testing

Which of the following best defines regression testing?

- Changes should be tested against all inputs that elicited bugs in earlier versions of the code.
- Every component in your code should have an associated set of tests that exercises all the corner cases in its specification.

You are not logged in.

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

- Tests should be written before you write the code as a way of checking your understanding of the specification.

- When a new test exposes a bug, you should run it on all previous versions of the code until you find the version where the bug was introduced.

CHECK

EXPLAIN

Running automated tests

Which of the following are good times to rerun all your JUnit tests?

Before doing git add/commit/push

After rewriting a function to make it faster

When using a code coverage tool

After you think you fixed a bug

Pushing your code to git sends it to the rest of your team, so rerun the tests first to make sure you're not pushing broken code.

Rewriting a function may introduce bugs, so rerun your tests to find them.

Rerunning tests is an essential part of using a code coverage tool, because you want to see the code lines that your tests don't reach.

Fixing a bug is a change to your program, and you should rerun your tests after every change.

CHECK

EXPLAIN

Testing techniques

Which of these techniques are useful for choosing test cases in test-first programming, before any code is written?

black box

regression

static typing

Reading 3: Testing

Validation

Why Software Testing is Hard

Putting on Your Testing Hat

Test-first Programming

Choosing Test Cases by Partitioning

Blackbox and Whitebox Testing

Coverage

Unit Testing and Stubs

Automated Testing and Regression Testing

Summary

An exercise for the reader

Collaboratively authored with contributions from:
Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama.

- partitioning
- boundaries
- white box
- coverage

- Regression testing creates test cases from bugs, so it only happens after (buggy) code has been written.
Static typing doesn't generate test cases.
White box testing and coverage both require looking at the written code.

CHECK

EXPLAIN

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.

An exercise for the reader

At this point you should have completed all the reading exercises above.

Reading 3: Testing

Validation

MIT EECS accessibility (<https://accessibility.mit.edu>)

Why Software Testing is

Hard

Putting on Your Testing

Hat

Test-first Programming

Choosing Test Cases by
Partitioning

Blackbox and Whitebox
Testing

Coverage

Unit Testing and Stubs

Automated Testing and
Regression Testing

Summary

An exercise for the
reader

Completing the reading exercises prepares you for the nanoquiz at the beginning of each class meeting, and

submitting the exercises is required by 10pm the evening before class.

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

Remember the exercises

Reading 4: Code Review

You must complete the **reading exercises** in this reading by 10:00 pm the night before class. Don't forget to log in using the big red **Log in** button. You will only receive credit for reading exercises if you are logged in when you do them.

Our Prime Objective in 6.005

Learning to write code that is:

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 (<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.

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

Remember the exercises

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 (<https://google-styleguide.googlecode.com/svn/trunk/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 (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 (<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.

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

Remember the exercises

```
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 (http://en.wikipedia.org/wiki/Don't_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?

READING EXERCISES

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

Remember the
exercises

Don't repeat yourself

Some of the repetition in `day0fYear()` is repeated values. How many times is the number of days in April written in `day0fYear()`?



s

8



Each sum of the form $31 + 28 + 31 + 30 + \dots$ is a sum of days in months:
 $31/*January*/ + 28/*February*/ + 31/*March*/ + 30/*April*/ + \dots$ There are 8 occurrences of 30 that belong to April.

By the way, the fact that this question couldn't be obviously answered from the code is an example of the problem of *magic numbers*, which will be discussed more in a bit.

CHECK

EXPLAIN

Don't repeat yourself

One reason why repeated code is bad is because a problem in the repeated code has to be fixed in many places, not just one. Suppose our calendar changed so that February really has 30 days instead of 28. How many numbers in this code have to be changed?



asd

10



The eight explicit occurrences of 28 would have to change, and so would the two numbers 59 and 90, which implicitly depend on the assumption that February has 28 days: $59 = 31/*January*/ + 28/*February*/$, and $90 = 31/*January*/ + 28/*February*/ + 31/*March*/$. These two surprise numbers are *magic numbers*, which we'll talk about shortly.

CHECK

EXPLAIN

Don't repeat yourself

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.

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

Remember the exercises

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 (<http://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:

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

Remember the exercises

```
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 `day0fYear` 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?

READING EXERCISES

Comments where needed

Which comments are useful additions to the code? Consider each comment independently, as if the other comments weren't there.

```
/** @param month month of the year, where January=1 and December=12 [C1] */
public static int day0fYear(int month, int day0fMonth, int year) {
    if (month == 2) {           // we're in February [C2]
        day0fMonth += 31;      // add in the days of January that already passed
    [C3]
    } else if (month == 3) {
        day0fMonth += 59;      // month is 3 here [C4]
    } else if (month == 4) {
        day0fMonth += 90;
    }
    ...
    } else if (month == 12) {
        day0fMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return day0fMonth; // the answer [C5]
}
```

C1

C2

- C3
- C4
- C5

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

Remember the exercises

➤ C1 is definitely a good addition because it clarifies what the month parameter means. In general, you should have a specification comment like this before every method. This specification isn't complete, but it's a start.

C2 and C3 help clarify what the numbers 2 and 31 signify. The comments help here, but we'll see shortly that a better way to explain these lines is not with comments but with descriptive names, like `month == FEBRUARY` and `dayOfMonth += MONTH_LENGTH[JANUARY]`.

C4 and C5 contribute nothing that a capable reader of Java wouldn't already know.

CHECK

EXPLAIN

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 fails faster than dynamic checking, and dynamic checking fails faster than producing a wrong answer that may corrupt subsequent computation.

The `dayOfYear` 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 `dayOfYear` 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.

READING EXERCISES

Fail fast

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

Remember the exercises

```
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;  
}
```

Suppose the date is February 9, 2019. The correct `dayOfYear()` result for this date is 40, since it's the fortieth day of the year.

Which of the following are plausible ways that a programmer might (mistakenly) call `dayOfYear()`? And for each one, does it lead to a static error, dynamic error, or wrong answer?

`dayOfYear(2, 9, 2019)`



plausible mistake – dynamic error



not a mistake – right answer



`dayOfYear()` expects its arguments in month/day/year order, with the month as a number from 1 to 12, so this is the right way to call it, and it will produce the right answer, 40.

`dayOfYear(1, 9, 2019)`

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

Remember the exercises



plausible mistake – static error



plausible mistake – wrong answer



This is plausible if the programmer is assuming zero-based indexing for months, i.e. the month is a number from 0 to 11, so February is 1. Zero-based indexing is far more common in programming than one-based indexing, so this is a very plausible mistake, but it quietly produces the wrong answer.

```
dayOfYear(9, 2, 2019)
```



plausible mistake – dynamic error



plausible mistake – wrong answer



This is plausible if the programmer is assuming the arguments are in day/month/year order, which is the standard almost everywhere in the world except the United States. It quietly produces the wrong answer because dayOfYear() interprets those arguments as September 2.

```
dayOfYear("February", 9, 2019)
```



plausible mistake – static error



plausible mistake – static error



This is plausible if the programmer is assuming the month is passed by a string name (in English). Static type checking forbids passing a String to an int argument, however, so the mistake is caught fast, before the program even starts.

```
dayOfYear(2019, 2, 9)
```



plausible mistake – wrong answer



plausible mistake – wrong answer



This is plausible if the programmer is assuming the arguments are in year/month/day order, which is a common international standard (ISO 8601, in fact). It quietly produces the wrong answer because none of the if statements match the huge month number, so it ends up treating it the same as January 2.

```
dayOfYear(2, 2019, 9)
```



plausible mistake – wrong answer



implausible mistake

- This is implausible because no convention for writing dates puts the year in the middle. It's unlikely to happen by accident.

CHECK

EXPLAIN

Reading 4: Code Review

Code Review

Smelly Example #1

Fail faster

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

Remember the exercises

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.)

Other constant numbers need to be explained. One way to explain them is with a comment, but a far better way is to declare the number as a constant with a good, explanatory 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.

READING EXERCISES

Avoid magic numbers

In the code:

```
if (month == 2) { ... }
```

what might a reasonable programmer plausibly assume about the meaning of the magic number 2?

- ✗ 2 might mean January
- 2 might mean February
- 2 might mean March

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

Remember the exercises

2 might mean the year 2 AD

- For one-based indexing (January=1), 2 would mean February, but for zero-based indexing (January=0), it would mean March.

CHECK

EXPLAIN

What happens when you assume

Suppose you're reading some code that uses a turtle graphics library that you don't know well, and you see the code:

```
turtle.rotate(3);
```

Which of the following are likely assumptions you might make about the meaning of the magic number 3?

- ✗ 3 might mean 3 degrees clockwise
- 3 might mean 3 degrees counterclockwise
- 3 might mean 3 radians clockwise
- 3 might mean 3 full revolutions

- All of these are reasonable assumptions for a magic number 3.

CHECK

EXPLAIN

Names instead of numbers

Consider this code:

```
for (int i = 0; i < 5; ++i) {  
    turtle.forward(36);  
    turtle.turn(72);  
}
```

The magic numbers in this code cause it to fail all three of our measures of code quality: it's not safe from bugs (SFB), not easy to understand (ETU) and not ready for change (RFC).

For each of the following rewrites, judge whether it improves SFB, ETU, and/or RFC, or none of the above.

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

Remember the exercises

```
final int FIVE = 5;
final int THIRTY_SIX = 36;
final int SEVENTY_TWO = 72;
for (int i = 0; i < FIVE; ++i) {
    turtle.forward(THIRTY_SIX);
    turtle.turn(SEVENTY_TWO);
}
```

✖ no improvement (or worse)

safer from bugs

easier to understand

more ready for change

```
int[] numbers = new int[] { 5, 36, 72 };
for (int i = 0; i < numbers[0]; ++i) {
    turtle.forward(numbers[1]);
    turtle.turn(numbers[2]);
}
```

✖ no improvement (or worse)

safer from bugs

easier to understand

more ready for change

```
int x = 5;
for (int i = 0; i < x; ++i) {
    turtle.forward(36);
    turtle.turn(360.0 / x);
}
```

✖ no improvement (or worse)

safer from bugs

easier to understand

more ready for change

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

Remember the exercises

- This is not particularly easier to understand, but at least it makes the relationship between the two magic numbers 5 and 72 more obvious. This makes the code safe from a potential bug: a future programmer can't change the number of iterations of the loop without also changing the turtle's rotation. This also makes the code more ready for change, because only one number needs to be edited instead of two.

```
final double FULL_CIRCLE_DEGREES = 360.0;
final int NUM_SIDES = 5;
final int SIDE_LENGTH = 36;
for (int i = 0; i < NUM_SIDES; ++i) {
    turtle.forward(SIDE_LENGTH);
    turtle.turn(FULL_CIRCLE_DEGREES / NUM_SIDES);
}
```

- ✖ no improvement (or worse)
 safer from bugs
 easier to understand
 more ready for change

- Contrast this version of the code with the original. Can you tell what it does? Notice that comments weren't needed to make the code clearer, just descriptive names.

CHECK

EXPLAIN

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 dayOfYear(final int month, final int dayOfMonth, final int year) {  
    ... }
```

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

Remember the exercises

Smelly Example #2

There was a latent bug in `dayOfYear`. 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.char  
At(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?

READING EXERCISES

Mental execution 2016

What happens when you call:

```
leap(2016)
```

✓ returns true on line R1 ✅

returns false on line R2

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

Remember the exercises

- returns false on line R3
- returns true on line R4
- returns false on line R5
- error before program starts
- error while program is running

➤ Here is how the code executes: `String.valueOf(2016)` returns the string "2016" and stores it in `tmp`. `tmp.charAt(2)` is the character '1', so the code enters the first `if` statement. `tmp.charAt(3)` is the character '6', so the return statement marked R1 is executed.

This is correct behavior. 2016 is in fact a leap year, so `leap(2016)` is behaving properly.

Mental execution of code is a crucial part of reading and understanding code, and is often necessary in code review to help find bugs.

CHECK

EXPLAIN

Mental execution 2017

What happens when you call:

`leap(2017)`

- ✓
- returns true on line R1
 - returns false on line R2
 - returns false on line R3
 - returns true on line R4
 - returns false on line 15
 - error before program starts
 - error while program is running

CHECK

EXPLAIN

Mental execution 2050

What happens when you call:

leap(2050)

- ✗ returns true on line R1
 returns false on line R2
 returns false on line R3
 returns true on line R4
 returns false on line R5
 error before program starts
 error while program is running

➤ Note that leap() is returning the wrong answer in this case, because 2050 is *not* a leap year.

The reason this happens is a subtle bug in the program: the expression `tmp.charAt(2) == 5`, which should be `tmp.charAt(2) == '5'`. Instead of testing the third character in 2050 against '5' and finding a match, it tests the character against the *integer* 5. Since the character's value is actually equivalent to the integer 53 (the Unicode value of the character '5'), the test fails, and the code goes on to a different part that produces the wrong answer.

You might ask why Java even allows a character to be compared against an integer – why isn't it a static type error? Indeed, it should be. If you tried to compare a `String` with an integer, Java would indeed produce a type error. But for historical reasons, coming from its legacy in the C and C++ programming languages, characters in Java are numeric types, just like `int` and `long`, and automatically convert to integers when they need to. Static typing didn't save us here, because Java's type system is very weak in places, like this automatic numeric type conversion.

We can also attribute this failure to the code's lack of DRYness. It contains twelve different character comparisons (`charAt(k) == 'c'` for some k or c), and all that repetitive code means that each one is a potential place for a bug. Just because the code works for 2016 doesn't give us much confidence that it will work for 2050, because there are so many different paths through this function.

CHECK

EXPLAIN

What happens when you call:

```
leap(10016)
```

- ✗ returns true on line R1
 returns false on line R2
 returns false on line R3
 returns true on line R4
 returns false on line R5
 error before program starts
 error while program is running

➤ This code is failing because it only really works on 4-digit years. It treats 10016 like the year 1001. Notice that this code doesn't fail fast – when given unexpected input, it simply produces the wrong answer.

CHECK

EXPLAIN

Mental execution 916

What happens when you call:

```
leap(916)
```

- ✗ returns true on line R1
 returns false on line R2
 returns false on line R3
 returns true on line R4
 returns false on line R5
 error before program starts
 error while program is running

➤ This code is making the assumption that the year has exactly 4 digits. When it has fewer than 4 digits, then `tmp.charAt(3)`, which tries to look at the fourth digit in the string will fail with a dynamic error, in the same way that indexing beyond the end of a Python string would fail with an error.

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

Remember the exercises

CHECK**EXPLAIN**

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

Remember the exercises

Magic numbers

How many magic numbers are in this code? Count every occurrence if some appear more than once.



jb

24



Every expression of the form `tmp.charAt(k) == 'n'` has two magic numbers in it, `k` and `n`. There are twelve such expressions.

CHECK**EXPLAIN**

DRYing out

Suppose you wrote the helper function:

```
public static boolean isDivisibleBy(int number, int factor) { return number %  
    factor == 0; }
```

If `leap()` were rewritten to use `isDivisibleBy(year, ...)`, and to correctly follow the leap year algorithm (http://en.wikipedia.org/wiki/Leap_year#Algorithm), how many magic numbers would be in the code?



hj

3



The three remaining magic numbers would be 4, 100, and 400. The rewritten method might look like:

```
public static int isLeapYear(int year) {  
    if (isDivisibleBy(year, 400)) return true;  
    else if (isDivisibleBy(year, 100)) return false;  
    else return isDivisibleBy(year, 4);  
}
```

In a sense these numbers truly are irreducible magic. They are arbitrary corrections, originally based on the ratio of the solar year to the solar day, so it's hard to give meaningful names to the values 4, 100, and 400. The three numbers are not easy to compute from first principles,

either, because they were determined by a standards body, not by an algorithm. As long as these three magic numbers are localized within `isLeapYear`, and not repeated anywhere else in the program, a software engineer would probably tolerate them.

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

Remember the exercises

CHECK

EXPLAIN

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?

READING EXERCISES

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

Remember the
exercises

Better method names

Which of the following are good names for the `leap()` method?



- leap
- isLeapYear
- IsLeapYear
- is_divisible_by_4



`IsLeapYear` might be a fine name in another programming language, but in Java convention, only classes are Capitalized.

`is_divisible_by_4` not only violates the Java convention (camelCase), but is also too low-level, and in fact inaccurate, since leap year computation is more complicated than divisibility by 4.

CHECK

EXPLAIN

Better variable names

Which of the following are good names for the `tmp` variable inside `leap()` ?



- leapYearString
- yearString
- temp
- secondsPerDay
- s



Out of these choices, `yearString` is the best.

`leapYearString` is inaccurate, because the year may not be a leap year after all. It's premature to call it one.

`temp` is just as bad as `tmp`.

`secondsPerDay` is completely inaccurate. Copy-and-pasting code can often produce out-of-context names like this, which is another reason to avoid copy and paste.

s is a popular name for a string variable when it really is generic, any string, with no more specific meaning than that. It's a poor choice here because this string is related to the year variable, so it's more specific and deserves a specific name.

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

Remember the exercises

CHECK

EXPLAIN

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

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

Remember the
exercises

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 (<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.

READING EXERCISES

Identifying global variables

In this code, which of these are global variables?

- ✖ countLongWords
 n
 LONG_WORD_LENGTH ✅
 longestWord ✅
 word
 words

➤ Recall that a global variable must be a variable, and it must be accessible from anywhere in the program.

`countLongWords` is a method, not a variable.

`n`, `word`, and `words` are local variables – their declarations are inside a method, which makes them accessible only within that method.

`LONG_WORD_LENGTH` and `longestWord` are declared outside the method. They also have the modifier “`public`”, which we'll discuss in more detail in a future class, but which means that even code outside this class can access and reassign those variables. Variables marked “`public static`” (but without `final`) are truly Java's global variables, and using them is always a bad idea.

You are not
logged in.

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

Remember the exercises

Note that `LONG_WORD_LENGTH` is actually named in a misleading way. By Java convention, names that are all uppercase letters should be constants, but unfortunately the keyword “final” was omitted from `LONG_WORD_LENGTH`’s declaration, which means it can be reassigned just like any other variable. It isn’t actually a constant.

CHECK

EXPLAIN

Effect of final

Making a variable into a constant by adding the `final` keyword can eliminate the risk of global variables. What happens to each of these when the `final` keyword is added?

`n`

✗ | becomes constant ▾ error before program starts

➤ With `final`, the compiler will notice that the code is trying to increment `n`. `“++n”` is equivalent to `“n += 1”`, which is equivalent to `“n = n+1”`, which reassigns `n`. You’ll get an error before the program starts.

`LONG_WORD_LENGTH`

✗ | error before program starts ▾ becomes constant

➤ With `final`, `LONG_WORD_LENGTH` will become constant. It is assigned once, when it is first declared, and never changed again in this code. Now it truly deserves to be written in all uppercase letters, because it is truly constant.

`longestWord`

✓ | error before program starts ▾ error before program starts

➤ With `final`, the compiler will notice that `longestWord` is reassigned in the body of the `for` loop, and it will signal a static error, just as it would for `n`. `n` and `longestWord` have to remain variables, but at least they aren’t global.

`word`

✗ | doesn't become constant ▾ becomes constant

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

Remember the exercises

With `final`, the word variable now can't be reassigned within the body of the for loop. Each time around the for loop, however, it will get a fresh value.

words

 error before program starts ▾ doesn't become constant

Adding `final` to the `words` variable means that the variable can't be reassigned – but the `List` object that it points to can still be mutated. This subtle difference can be a source of bugs, and we'll see it again and again this semester.

CHECK

EXPLAIN

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:

Reading 4: Code Review

Code Review

Smelly Example #1

Don't Repeat Yourself

Comments Where Needed

Fail Fast

Avoid Magic Numbers

Collaboratively authored with contributions from:

One Purpose for Each Variable

Chiplala, Srinivas Devadas, Michael Ernst, Max

Goldman, John Guttag, Daniel Jackson, Rob Miller,

Martin Rinard, and

Use Good Names.

This work is licensed under

Use Whitespace to Help the Reader

(<http://creativecommons.org/licenses/by-sa/4.0/>).

Smelly Example #2

Don't Use Global Variables

Methods Should Return Results, not Print Them

Summary

Remember the exercises

- **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.

Remember the exercises

At this point you should have completed all the reading exercises above.

Completing the reading exercises prepares you for the *nanoquiz* at the beginning of each class meeting, and submitting the exercises is required by 10pm the evening before class.

MIT EECS accessibility (<https://accessibility.mit.edu>)

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:

Reading 6:
Specifications

Introduction

Part 1: Specifications

Part 2: Exceptions

Summary

READING EXERCISES

Scrabble 1

```
/**  
 * Requires: tiles has length 7 & contains only uppercase letters.  
 *           crossings contains only uppercase letters, without duplicates.  
 * Effects: Returns a list of words where each word can be made by taking  
 *           letters from tiles and at most 1 letter from crossings.  
 */  
public static List<String> scrabble(String tiles, String crossings) {  
    if (tiles.length() != 7) { throw new RuntimeException(); }  
    return new ArrayList<>();  
}
```

You are not
logged in.

Which are parts of the *postcondition* of scrabble ?

- ✓ tiles has only uppercase letters
- crossings has no duplicates
- scrabble takes two arguments
- scrabble returns a list of strings 

➤ The first three answers are *preconditions*.

Which are parts of the *precondition* of scrabble ?

- ✗ tiles has length 7 
- crossings is a string of uppercase letters 
- scrabble 's arguments are of type String and String 
- scrabble returns an empty ArrayList

➤ The last answer is true about the implementation but it is not part of the spec.

CHECK

EXPLAIN

Scrabble 2

Reading 6:
Specifications

Introduction

Part 1: Specifications

Part 2: Exceptions

Summary

Which are *parts of the spec* that are *checked statically* by Java?

- ✗ tiles is a string of uppercase letters
 crossings has no duplicates
 when tiles.length() != 7, scrabble throws a RuntimeException
 scrabble takes two arguments ✓

➤ The first two are preconditions that are not expressed by the types, Java cannot check them statically. The third answer is true about the implementation but it is not part of the spec, and it also cannot be checked statically.

Does the scrabble implementation satisfy its specification?

- ✓ Yes ✓
 No, because it throws a RuntimeException when tiles has invalid length
 No, because it returns an empty result list even if we call it with tiles and crossings that can be combined to form valid words

➤ The specification doesn't say that the method returns *all* words that can be made from the inputs, or even *some* words that can be made.

This is a clear place where the specification could be improved.

CHECK

EXPLAIN

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.

Collaboratively authored
with contributions from:
Saman Amarasinghe, Adam
Chlipala, Srini Devadas,
Michael Ernst, Max

- **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.

Introduction

Part 1: Specifications

MIT EECS accessibility (<https://accessibility.mit.edu>)

Part 2: Exceptions

Summary

Reading 6, Part 1: Specifications

[Why specifications?](#)[Behavioral equivalence](#)[Specification structure](#)[Null references](#)[What a specification may talk about](#)[Testing and specifications](#)[Specifications for mutating methods](#)[Next: Exceptions](#)

Reading 6, Part 1: Specifications

Before we dive into the structure and meaning of specifications...

Why specifications?

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

Specifications are good for the client of a method because they spare the task of reading code. If you're not convinced that reading a spec is easier than reading code, take a look at some of the standard Java specs and compare them to the source code that implements them.

Here's an example of one method from `BigInteger` (<http://docs.oracle.com/javase/8/docs/api/?java/math/BigInteger.html>) — a class for representing integers up to arbitrary size without the size limit of primitive `int` — next to its code:

Specification from the API documentation

(<http://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html#add-java.math.BigInteger->):

```
add

public BigInteger add(BigInteger val)
Returns a BigInteger whose value is (this + val) .

Parameters:
val - value to be added to this BigInteger.

Returns:
this + val
```

Method body from Java 8 source

(<http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java/math/BigInteger.java#l1207>):

```
if (val.signum == 0)
    return this;
if (signum == 0)
    return val;
if (val.signum == signum)
    return new BigInteger(add(mag, val.mag), signum);

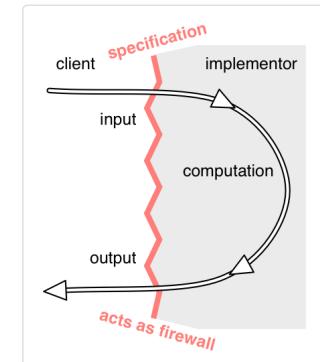
int cmp = compareMagnitude(val);
if (cmp == 0)
    return ZERO;
int[] resultMag = (cmp > 0 ? subtract(mag, val.mag)
                           : subtract(val.mag, mag));
resultMag = trustedStripLeadingZeroInts(resultMag);

return new BigInteger(resultMag, cmp == signum ? 1 : -1);
```

The spec for `BigInteger.add` is straightforward for clients to understand, and if we have questions about corner cases, the `BigInteger` class provides additional human-readable documentation. If all we had was the code, we'd have to read through the `BigInteger` constructor, `compareMagnitude`, `subtract`, and `trustedStripLeadingZeroInts` just as a starting point.

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

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



Behavioral equivalence

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

Reading 6, Part 1: Specifications

Why specifications?

Behavioral equivalence

Specification structure

Null references

What a specification may talk about

Testing and specifications

Specifications for mutating methods

Next: Exceptions

```
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;
}
```

Of course the code is different, so in that sense they are different; and we've given them different names, just for the purpose of discussion. To determine *behavior equivalence*, our question is whether we could substitute one implementation for the other.

Not only do these methods have different code, they actually have different behavior:

- when `val` is missing, `findFirst` returns the length of `arr` and `findLast` returns `-1`;
- when `val` appears twice, `findFirst` returns the lower index and `findLast` returns the higher.

But when `val` occurs at exactly one index of the array, the two methods behave the same: they both return that index. It may be that clients never rely on the behavior in the other cases. Whenever they call the method, they will be passing in an `arr` with exactly one element `val`. For such clients, these two methods are the same, and we could switch from one implementation to the other without issue.

The notion of equivalence is in the eye of the beholder — that is, the client. In order to make it possible to substitute one implementation for another, and to know when this is acceptable, we need a specification that states exactly what the client depends on.

In this case, our specification might be:

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

READING EXERCISES

Behave nicely

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

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

As we said above, suppose clients only care about calling the `find` method when they know `val` occurs exactly once in `a`.

In this case, are `findFirst` and `findLast` behaviorally equivalent?

- ✓ Yes ✎
 No

➤ If `val` occurs exactly once in `a`, then it doesn't matter whether we search from start to end or end to start; and it doesn't matter what we do when we don't find `val`.

Once we define how specifications are structured, we'll see that they are equivalent in this case because a strong precondition hides their potential differences in behavior.

Reading 6, Part 1: Specifications

[CHECK](#) [EXPLAIN](#)

Why specifications?

Behavioral equivalence

Specification structure

Null references

What a specification may talk about

Testing and specifications

Specifications for mutating methods

Next: Exceptions

Best behavior

Now let's change the spec.

Suppose clients only care that the find method should return:

- any index i such that $a[i] == val$, if val is in a
- any integer j such that j is not a valid array index, otherwise

In this case, are `findFirst` and `findLast` behaviorally equivalent?

Yes

No

- Both implementations satisfy these very minimal requirements on the return value.

Once we define how specifications are structured, we'll see that they are equivalent in this case because a weak postcondition permits their differences in behavior.

[CHECK](#) [EXPLAIN](#)

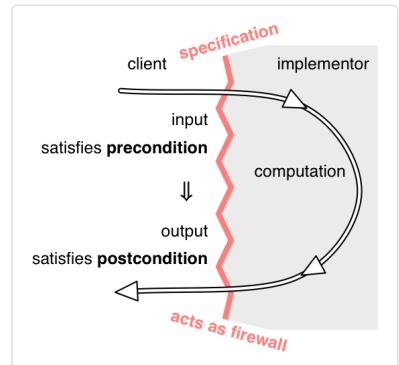
Specification structure

A specification of a method consists of several clauses:

- a *precondition*, indicated by the keyword *requires*
- a *postcondition*, indicated by the keyword *effects*

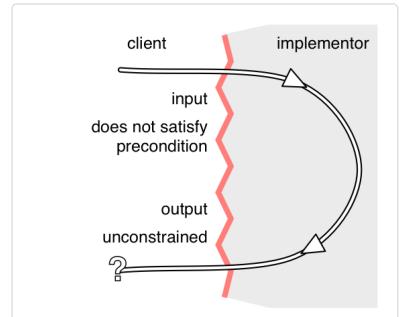
The precondition is an obligation on the client (i.e., the caller of the method). It's a condition over the state in which the method is invoked.

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



The overall structure is a logical implication: *if* the precondition holds when the method is called, *then* the postcondition must hold when the method completes.

If the precondition does *not* hold when the method is called, the implementation is *not* bound by the postcondition. It is free to do anything, including not terminating, throwing an exception, returning arbitrary results, making arbitrary modifications, etc.



READING EXERCISES

Logical implication

Here's the spec we've been looking at:

Reading 6, Part 1: Specifications

Why specifications?

Behavioral equivalence

Specification structure

Null references

What a specification may talk about

Testing and specifications

Specifications for mutating methods

Next: Exceptions

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

As the implementer of `find`, which are legal?

- if `arr` is empty, return 0
- if `arr` is empty, throw an exception
- if `val` occurs twice in `arr`, throw an exception
- if `val` occurs twice in `arr`, set all the values in `arr` to zero, then throw an exception
- if `arr` is not empty but `val` doesn't occur, pick an index at random, set it to `val`, and return that index
- if `arr[0]` is `val`, continue checking the rest of the array and return the highest index where you find `val` (or 0 if you don't find it again)

► In all but the last case, the client has *violated the precondition*. As implementors, we are free to do anything, including the unhelpful or the downright malicious.

In the last case, the precondition is satisfied, so we must satisfy the postcondition. The postcondition allows us to return *any* index `i`, so we are free to search unnecessarily for a larger one.

CHECK

EXPLAIN

Logical implementation

As the implementor of `find`, why would you choose to throw an exception if `arr` is empty?

- DRY
- fail fast
- avoid magic numbers
- one purpose per variable
- avoid global variables
- return results

► When our precondition is violated, the client has a bug. We can make that bug easier to find and fix by failing fast, even though we are not obligated to do so.

CHECK

EXPLAIN

Specifications in Java

Some languages (notably Eiffel ([http://en.wikipedia.org/wiki/Eiffel_\(programming_language\)](http://en.wikipedia.org/wiki/Eiffel_(programming_language)))) incorporate preconditions and postconditions as a fundamental part of the language, as expressions that the runtime system (or even the compiler) can automatically check to enforce the contracts between clients and implementers.

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

Java has a convention for documentation comments (<http://en.wikipedia.org/wiki/Javadoc>), in which parameters are described by `@param` clauses and results are described by `@return` and `@throws` clauses. You should put the preconditions into `@param` where possible, and postconditions into `@return` and `@throws`. So a specification like this:

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

... might be rendered in Java like this:

```

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

```

The Java API documentation (<http://docs.oracle.com/javase/8/docs/api/>) is produced from Javadoc comments in the Java standard library source code (<http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java>). Documenting your specifications in Javadoc allows Eclipse to show you (and clients of your code) useful information, and allows you to produce HTML documentation (<http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fref-export-javadoc.htm>) in the same format as the Java API docs.

Read: *Introduction, Commenting in Java, and Javadoc Comments in Javadoc Comments* (<http://javaworkshop.sourceforge.net/chapter4.html>).

When writing your specifications, you can also refer to Oracle's **How to Write Doc Comments** (<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>).

READING EXERCISES

Javadoc

You are not logged in.

Given this spec:

```

static boolean isPalindrome(String word)
    requires: word contains only alphanumeric characters
    effects: returns true if and only if word is a palindrome

```

Which pieces should we include in the Javadoc?

- /*
- * Check if a word is a palindrome. ↗
- * A palindrome is a sequence of characters
* that reads the same forwards and backwards. ↗
- * @param String word
- * @requires word contains only alphanumeric characters
- * @effects returns true if and only if word is a palindrome
- * @return boolean
- * @throws nothing
- */

Here's one possible Javadoc comment:

```

/**
 * Check if a word is a palindrome.
 * A palindrome is a sequence of characters
 * that reads the same forwards and backwards.
 * @param word word to check, must contain only alphanumeric characters
 * @return true if and only if word is a palindrome
 */

```

Review the pair of pages linked above for the correct usage of each tag.

CHECK

EXPLAIN

Null references

In Java, references to objects and arrays can also take on the special value `null`, which means that the reference doesn't point to an object. Null values are an unfortunate hole in Java's type system.

Reading 6, Part 1: Specifications

Why specifications?

Behavioral equivalence

Specification structure

Null references

What a specification may talk about

Testing and specifications

Specifications for mutating methods

Next: Exceptions

```
int size = null; // illegal  
double depth = null; // illegal
```

and the compiler will reject such attempts with static errors.

On the other hand, we can assign `null` to any non-primitive variable:

```
String name = null;  
int[] points = null;
```

and the compiler happily accepts this code at compile time. But you'll get errors at runtime because you can't call any methods or use any fields with one of these references:

```
name.length() // throws NullPointerException  
points.length // throws NullPointerException
```

Note, in particular, that `null` is not the same as an empty string `""` or an empty array. On an empty string or empty array, you *can* call methods and access fields. The length of an empty array or an empty string is 0. The length of a string variable that points to `null` throws a `NullPointerException`.

Null values are troublesome and unsafe, so much so that you're well advised to remove them from your design vocabulary. In 6.005 — and in fact in most good Java programming — **null values are implicitly disallowed as parameters and return values**. So every method implicitly has a precondition on its object and array parameters that they be non-null. Every method that returns an object or an array implicitly has a postcondition that its return value is non-null. If a method allows null values for a parameter, it should explicitly state it, or if it might return a null value as a result, it should explicitly state it. But these are in general not good ideas. **Avoid null**.

There are extensions to Java that allow you to forbid `null` directly in the type declaration, e.g.:

```
static boolean addAll(@NonNull List<T> list1, @NonNull List<T> list2)
```

where it can be checked automatically (<http://types.cs.washington.edu/checker-framework/>) at compile time or runtime.

Google has their own discussion of `null` in Guava, the company's core Java libraries (<https://code.google.com/p/guava-libraries/wiki/UsingAndAvoidingNullExplained>). The project explains:

Careless use of `null` can cause a staggering variety of bugs. Studying the Google code base, we found that something like 95% of collections weren't supposed to have any null values in them, and having those **fail fast** rather than silently accept `null` would have been helpful to developers.

Additionally, `null` is unpleasantly ambiguous. It's rarely obvious what a `null` return value is supposed to mean — for example, `Map.get(key)` can return `null` either because the value in the map is `null`, or the value is not in the map. Null can mean failure, can mean success, can mean almost anything. Using something other than `null` **makes your meaning clear**.

(Emphasis added.)

READING EXERCISES

NullPointerException accessing exercise.name()

Which of the following can be null?

If you're not sure, try it yourself in a small Java program

([http://www.pythontutor.com/java.html#code=import+java.util.*%3B%0A%0Apublic+class+WhatCanBeNull+%7B%0A++++public+static+void+main\(String%5B%5D+args%29+%7B%0A++++++int+a=%3D+null%0A++++++System.out.println\(a\);%7D%0A%0A%7D](http://www.pythontutor.com/java.html#code=import+java.util.*%3B%0A%0Apublic+class+WhatCanBeNull+%7B%0A++++public+static+void+main(String%5B%5D+args%29+%7B%0A++++++int+a=%3D+null%0A++++++System.out.println(a);%7D%0A%0A%7D))

Check all that apply:

- int a;
- char b;
- double c;
- int[] d;
- String e;

Reading 6, Part 1: Specifications

Why specifications?

Behavioral equivalence

Specification structure

Null references

What a specification may talk about

Testing and specifications

Specifications for mutating methods

Next: Exceptions

- String[] f;
- Double g;
- List<Integer> h;
- final MouseTrap i;
- static final String j;

➤ All array and object type variables, including the primitive wrappers, can be `null`. `List<Integer> h` can both be `null` and it can contain `null` as entries in the list.

Primitive type variables cannot be `null`.

CHECK

EXPLAIN

There are null exercises remaining

```
public static String none() {  
    return null;          // (1)  
}  
  
public static void main(String[] args) {  
    String a = none();    // (2)  
    String b = null;      // (3)  
    if (a.length() > 0) { // (4)  
        b = a;            // (5)  
    }  
    return b;             // (6)  
}
```

Which line contains a static error?

4 6

➤ `null` is not the same as `void`, and a `void` method like `main` cannot try to return it.

If we comment out that line and run `main` ...

Which line contains a dynamic error?

2 4

➤ Since `none()` returns `null`, `a` will be `null`, and calling `length()` will cause a `NullPointerException`.

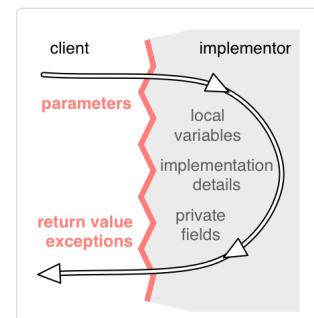
CHECK

EXPLAIN

What a specification may talk about

A specification of a method can talk about the parameters and return value of the method, but it should never talk about local variables of the method or private fields of the method's class. You should consider the implementation invisible to the reader of the spec.

In Java, the source code of the method is often unavailable to the reader of your spec, because the Javadoc tool extracts the spec comments from your code and renders them as HTML.



Testing and specifications

Reading 6, Part 1: Specifications

Why specifications?

Behavioral equivalence

Specification structure

Null references

What a specification may talk about

Testing and specifications

Specifications for mutating methods

Next: Exceptions

In testing, we talk about *black box tests* that are chosen with only the specification in mind, and *glass box tests* that are chosen with knowledge of the actual implementation (*Testing* (http://web.mit.edu/6.005/www/fa15/classes/03-testing/#blackbox_and_whitebox_testing)). But it's important to note that **even glass box tests must follow the specification**. Your implementation may provide stronger guarantees than the specification calls for, or it may have specific behavior where the specification is undefined. But your test cases should not count on that behavior. Test cases must obey the contract, just like every other client.

For example, suppose you are testing this specification of `find`, slightly different from the one we've used so far:

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

This spec has a strong precondition in the sense that `val` is required to be found; and it has a fairly weak postcondition in the sense that if `val` appears more than once in the array, this specification says nothing about which particular index of `val` is returned. Even if you implemented `find` so that it always returns the lowest index, your test case can't assume that specific behavior:

```
int[] array = new int[] { 7, 7, 7 };
assertEquals(0, find(array, 7)); // bad test case: violates the spec
assertEquals(7, array[find(array, 7)]); // correct
```

Similarly, even if you implemented `find` so that it (sensibly) throws an exception when `val` isn't found, instead of returning some arbitrary misleading index, your test case can't assume that behavior, because it can't call `find()` in a way that violates the precondition.

So what does glass box testing mean, if it can't go beyond the spec? It means you are trying to find new test cases that exercise different parts of the implementation, but still checking those test cases in an implementation-independent way.

Testing units

Recall the web search example from *Testing* (http://web.mit.edu/6.005/www/fa15/classes/03-testing/#unit_testing_and_stubs) with these methods:

```
/** @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) { ... }

/** @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) {
    ...
    calls getWebPage and extractWords
    ...
}
```

We talked then about *unit testing*, the idea that we should write tests of each module of our program in isolation. A good unit test is focused on just a single specification. Our tests will nearly always rely on the specs of Java library methods, but a unit test for one method we've written shouldn't fail if a *different* method fails to satisfy its spec. As we saw in the example, a test for `extractWords` shouldn't fail if `getWebPage` doesn't satisfy its postcondition.

Good *integration tests*, tests that use a combination of modules, will make sure that our different methods have compatible specifications: callers and implementors of different methods are passing and returning values as the other expects. Integration tests cannot replace systematically-designed unit tests. From the example, if we only ever test `extractWords` by calling `makeIndex`, we will only test it on a potentially small part of its input space: inputs that are possible outputs of `getWebPage`. In doing so, we've left a place for bugs to hide, ready to jump out when we use `extractWords` for a different purpose elsewhere in our program, or when `getWebPage` starts returning web pages written in a new format, etc.

Specifications for mutating methods

We previously discussed mutable vs. immutable objects, but our specifications of `find` didn't give us the opportunity to illustrate how to describe side-effects — changes to mutable data — in the postcondition.

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

```

static boolean addAll(List<T> list1, List<T> list2)
  requires: list1 != list2
  effects: modifies list1 by adding the elements of list2 to the end of
           it, and returns true if list1 changed as a result of call

```

Reading 6, Part 1: Specifications

[Why specifications?](#)

[Behavioral equivalence](#)

[Specification structure](#)

[Null references](#)

[What a specification may talk about](#)

[Testing and specifications](#)

[Specifications for mutating methods](#)

[Next: Exceptions](#)

We've taken this, slightly simplified, from the Java `List` (<http://docs.oracle.com/javase/8/docs/api/?java/util/List.html>) interface. First, look at the postcondition. It gives two constraints: the first telling us how `list1` is modified, and the second telling us how the return value is determined.

Second, look at the precondition. It tells us that the behavior of the method if you attempt to add the elements of a list to itself is undefined. You can easily imagine why the implementor of the method would want to impose this constraint: it's not likely to rule out any useful applications of the method, and it makes it easier to implement. The specification allows a simple implementation in which you take an element from `list2` and add it to `list1`, then go on to the next element of `list2` until you get to the end. If `list1` and `list2` are the same list, this algorithm will not terminate — an outcome permitted by the specification because of its precondition.

Remember also our implicit precondition that `list1` and `list2` must be valid objects, rather than `null`. We'll usually omit saying this because it's virtually always required of object references.

Here is another 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()

```

Just as we've said that `null` is implicitly disallowed unless stated otherwise, we will also use the convention that **mutation is disallowed unless stated otherwise**. The spec of `toLowerCase` could explicitly state as an effect that "lst is not modified", but in the absence of a postcondition describing mutation, we demand no mutation of the inputs.

READING EXERCISES

What's in a spec?

Which of the following are part of a function's specification?

- 1. return type
- 2. restrictions on return values
- 3. number of arguments
- 4. argument types
- 5. restrictions on argument values

► 1 and 2 are postconditions; 3, 4, and 5 are preconditions.

1, 3, and 4 are statically checked by Java; 2 and 5 are usually not.

[CHECK](#)

[EXPLAIN](#)

gcd 1

Alice writes the following code:

Reading 6, Part 1: Specifications

Why specifications?

Behavioral equivalence

Specification structure

Null references

What a specification may talk about

Testing and specifications

Specifications for mutating methods

Next: Exceptions

```
public static int gcd(int a, int b) {
    if (a > b) {
        return gcd(a-b, b);
    } else if (b > a) {
        return gcd(a, b-a);
    }
    return a;
}
```

Bob writes the following test:

```
@Test public void gcdTest() {
    assertEquals(6, gcd(24, 54));
}
```

The test passes!

Alice should write $a > 0$ in the precondition of gcd

True False

Alice should write $b > 0$ in the precondition of gcd

True False

Alice should write $\text{gcd}(a, b) > 0$ in the precondition of gcd

True False

Alice should write a and b are integers in the precondition of gcd

True False

► The function requires $a > 0$ and $b > 0$ in order to return a correct answer.

$\text{gcd}(a, b) > 0$ is not a precondition, although it is a postcondition.

The compiler already checks that a and b are integers, so writing it again would be duplicative.

CHECK

EXPLAIN

gcd 2

If Alice adds $a > 0$ to the precondition, Bob should test negative values of a

True False

If Alice does not add $a > 0$ to the precondition, Bob should test negative values of a

True False

► Bob should not test inputs that violate the precondition. But without the precondition, a reasonable partitioning of input a would certainly include negative values.

CHECK

EXPLAIN

Next: Exceptions (./exceptions/)

Reading 6, Part 2:

Exceptions

Exceptions for signaling bugs

Now that we're writing specifications and thinking about how clients will use our methods, let's discuss how to handle *exceptional* cases in a way that is safe from bugs and easy to understand.

Exceptions for special results

A method's *signature* — its name, parameter types, return type — is a core part of its specification, and the signature may also include *exceptions* that the method may trigger.

Checked and unchecked exceptions

Throwable hierarchy

You've probably already seen some exceptions in your Java programming so far, such as `ArrayIndexOutOfBoundsException` (<http://docs.oracle.com/javase/8/docs/api/?java/lang/IndexOutOfBoundsException.html>) (thrown when an array index `foo[i]` is outside the valid range for the array `foo`) or `NullPointerException` (<http://docs.oracle.com/javase/8/docs/api/?java/lang/NullPointerException.html>) (thrown when trying to call a method on a `null` object reference).

These exceptions generally indicate **bugs** in your code, and the information displayed by Java when the exception is thrown can help you find and fix the bug.

`ArrayIndexOutOfBoundsException` and `NullPointerException` are probably the most common exceptions of this sort. Other examples include:

- `ArithmaticException` (<http://docs.oracle.com/javase/8/docs/api/?java/lang/ArithmaticException.html>), thrown for arithmetic errors like integer division by zero.
- `NumberFormatException` (<http://docs.oracle.com/javase/8/docs/api/?java/lang/NumberFormatException.html>), thrown by methods like `Integer.parseInt` if you pass in a string that cannot be parsed into an integer.

Exceptions for special results

Exceptions are not just for signaling bugs. They can be used to improve the structure of code that involves procedures with special results.

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

Reading 6, Part 2: Exceptions

Reading 6, Part 2:

Exceptions

Exceptions for signaling bugs

Now that we're writing specifications and thinking about how clients will use our methods, let's discuss how to handle *exceptional* cases in a way that is safe from bugs and easy to understand.

Exceptions for special results

A method's *signature* — its name, parameter types, return type — is a core part of its specification, and the signature may also include *exceptions* that the method may trigger.

Exceptions for signaling bugs

You've probably already seen some exceptions in your Java programming so far, such as `ArrayIndexOutOfBoundsException` (<http://docs.oracle.com/javase/8/docs/api/?java/lang/IndexOutOfBoundsException.html>) (thrown when an array index `foo[i]` is outside the valid range for the array `foo`) or `NullPointerException` (<http://docs.oracle.com/javase/8/docs/api/?java/lang/NullPointerException.html>) (thrown when trying to call a method on a `null` object reference).

These exceptions generally indicate **bugs** in your code, and the information displayed by Java when the exception is thrown can help you find and fix the bug.

`ArrayIndexOutOfBoundsException` and `NullPointerException` are probably the most common exceptions of this sort. Other examples include:

- `ArithmaticException` (<http://docs.oracle.com/javase/8/docs/api/?java/lang/ArithmaticException.html>), thrown for arithmetic errors like integer division by zero.
- `NumberFormatException` (<http://docs.oracle.com/javase/8/docs/api/?java/lang/NumberFormatException.html>), thrown by methods like `Integer.parseInt` if you pass in a string that cannot be parsed into an integer.

Exceptions for special results

Exceptions are not just for signaling bugs. They can be used to improve the structure of code that involves procedures with special results.

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

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

Reading 6, Part 2:

Exceptions

Exceptions for signaling bugs

Exceptions for special results

Checked and unchecked exceptions

Throwable hierarchy

Exception design considerations

Abuse of exceptions

Next: Summary

```
class BirthdayBook {  
    LocalDate lookup(String name) { ... }  
}
```

(`LocalDate` (<http://docs.oracle.com/javase/8/docs/api/?java/time/LocalDate.html>) is part of the Java API.)

What should the method do if the birthday book doesn't have an entry for the person whose name is given?

Well, we could return some special date that is not going to be used as a real date. Bad programmers have been doing this for decades; they would return 9/9/99, for example, since it was *obvious* that no program written in 1960 would still be running at the end of the century. (They were wrong, by the way.

(http://en.wikipedia.org/wiki/Year_2000_problem)

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

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

and the caller handles the exception with a `catch` clause. For example:

```
BirthdayBook birthdays = ...  
try {  
    LocalDate birthdate = birthdays.lookup("Alyssa");  
    // we know Alyssa's birthday  
} catch (NotFoundException nfe) {  
    // her birthday was not in the birthday book  
}
```

Now there's no need for any special value, nor the checking associated with it.

Read: **Exceptions** (<http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html>) in the Java Tutorials.

READING EXERCISES

1st birthday

Reading 6, Part 2: Exceptions

Exceptions for signaling bugs

Exceptions for special results

Checked and unchecked exceptions

Throwable hierarchy

Exception design considerations

Abuse of exceptions

Next: Summary

Assume we're using `BirthdayBook` with the `lookup` method that throws `NotFoundException`.

Assume we have initialized the `birthdays` variable to point to a `BirthdayBook`, and assume that Elliot is *not* in that birthday book.

What will happen with the following code:

```
try {  
    LocalDate birthdate = birthdays.lookup("Elliot");  
}
```

✖ static error: no such method or constructor ▾ static error: incorrect syntax

➤ We cannot have a `try` without a `catch` or `finally`.

CHECK

EXPLAIN

2nd birthday

```
try {  
    LocalDate birthdate = birthdays.lookup("Elliot");  
} catch (NotFoundException nfe) {  
    birthdate = LocalDate.now();  
}
```

✖ static error: incorrect syntax ▾ static error: undeclared variable

➤ Variables in `try - catch` blocks obey normal scoping rules. When `birthdate` is declared in the `try` block, it goes out of scope as soon as that block ends. We cannot assign to it, or refer to it at all, either in the `catch` or elsewhere.

CHECK

EXPLAIN

3rd birthday

Reading 6, Part 2: Exceptions

Exceptions for signaling
bugs

Exceptions for special
results

Checked and unchecked
exceptions

Throwable hierarchy

Exception design
considerations

Abuse of exceptions

Next: Summary

```
try {
    LocalDate birthdate = birthdays.lookup("Elliot");
} catch (NotFoundException nfe) {
    throw new DateTimeException("Missing reference birthday", nfe);
}
```

([DateTimeException](http://docs.oracle.com/javase/8/docs/api/java/time/DateTimeException.html) (<http://docs.oracle.com/javase/8/docs/api/?java/time/DateTimeException.html>) is provided by the Java API.)



static error: undeclared variable



dynamic error: DateTimeException



We catch the `NotFoundException` and throw a `DateTimeException` instead.

The pattern of **chained exceptions**

(<http://docs.oracle.com/javase/tutorial/essential/exceptions/chained.html>) is useful for throwing exceptions that are meaningful and useful to the client while preserving this underlying cause of the exception for debugging.

CHECK

EXPLAIN

Checked and unchecked exceptions

We've seen two different purposes for exceptions: special results and bug detection. As a general rule, you'll want to use checked exceptions to signal special results and unchecked exceptions to signal bugs. In a later section, we'll refine this a bit.

Some terminology: *checked* exceptions are called that because they are checked by the compiler:

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

So if you call `BirthdayBook`'s `lookup` method and forget to handle the `NotFoundException`, the compiler will reject your code. This is very useful, because it ensures that exceptions that are expected to occur will be handled.

Reading 6, Part 2: Exceptions

Exceptions for signaling bugs

Exceptions for special results

Checked and unchecked exceptions

Throwable hierarchy

Exception design considerations

Abuse of exceptions

Next: Summary

Unchecked exceptions, in contrast, are used to signal bugs. These exceptions are not expected to be handled by the code except perhaps at the top level. We wouldn't want every method up the call chain to have to declare that it (might) throw all the kinds of bug-related exceptions that can happen at lower call levels: index out of bounds, null pointers, illegal arguments, assertion failures, etc.

As a result, for an unchecked exception the compiler will not check for `try - catch` or a `throws` declaration. Java still allows you to write a `throws` clause for an unchecked exception as part of a method signature, but this has no effect, and is thus a bit funny, and we don't recommend doing it.

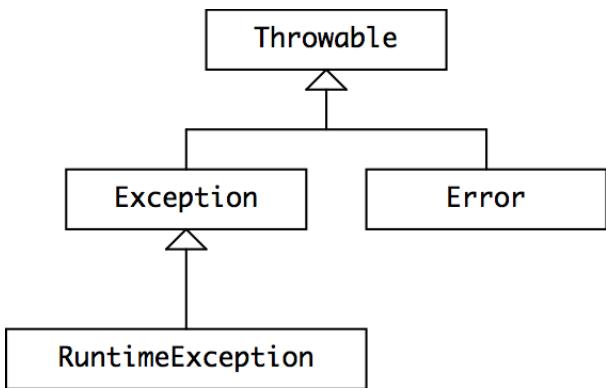
All exceptions may have a message associated with them. If not provided in the constructor, the reference to the message string is `null`.

Throwable hierarchy

To understand how Java decides whether an exception is checked or unchecked, let's look at the class hierarchy for Java exceptions.

`Throwable` (<http://docs.oracle.com/javase/8/docs/api/?java/lang/Throwable.html>) is the class of objects that can be thrown or caught. `Throwable`'s implementation records a stack trace at the point where the exception was thrown, along with an optional string describing the exception. Any object used in a `throw` or `catch` statement, or declared in the `throws` clause of a method, must be a subclass of `Throwable`.

`Error` (<http://docs.oracle.com/javase/8/docs/api/?java/lang/Error.html>) is a subclass of `Throwable` that is reserved for errors produced by the Java runtime system, such as `StackOverflowError` (<http://docs.oracle.com/javase/8/docs/api/?java/lang/StackOverflowError.html>) and `OutOfMemoryError` (<http://docs.oracle.com/javase/8/docs/api/?java/lang/OutOfMemoryError.html>). For some reason `AssertionError` (<http://docs.oracle.com/javase/8/docs/api/?java/lang/AssertionError.html>) also extends `Error`, even though it indicates a bug in user code, not in the runtime. Errors should be considered unrecoverable, and are generally not caught.



Here's how Java distinguishes between checked and unchecked exceptions:

- `RuntimeException`, `Error`, and their subclasses are **unchecked** exceptions. The compiler doesn't require them to be declared in the `throws` clause of a method that throws them, and doesn't require them to be caught or declared by a caller of such a method.
- All other throwables — `Throwable`, `Exception`, and all of their subclasses except for those of the `RuntimeException` and `Error` lineage — are **checked** exceptions. The compiler requires these exceptions to be caught or declared when it's possible for them to be thrown.

When you define your own exceptions, you should either subclass `RuntimeException` (to make it an unchecked exception) or `Exception` (to make it checked). Programmers generally don't subclass `Error` or `Throwable`, because these are reserved by Java itself.

Reading 6, Part 2:

Exceptions

Exceptions for signaling bugs

Exceptions for special results

Checked and unchecked exceptions

Throwable hierarchy

Exception design considerations

Abuse of exceptions

Next: Summary

READING EXERCISES

Get to the point

Suppose we're building a robot and we want to specify the function

```
public static List<Point> findPath(Point initial, Point goal)
```

which is responsible for path-finding: determining a sequence of `Point`s that the robot should move through to navigate from `initial` to `goal`, past any obstacles that might be in the way.

In the postcondition, we say that `findPath` will search for paths only up to a bounded length (set elsewhere), and that it will throw an exception if it fails to find one.

Which of these would we choose?

- ✖ throws a checked `NoPathException`
 throws an unchecked `NoPathException`
 throws a checked `PathNotFoundException` ✅
 throws an unchecked `PathNotFoundException`

➤ `NoPathException` is a bad name, since there might be very long paths the function doesn't find.

Unless we're working in a domain where not having a path signals a bug, it seems like not finding a path is a condition that the client must anticipate.

And it's not feasible for the client to check beforehand whether there is a path (that's what this function does)... so a *checked* exception is appropriate.

CHECK

EXPLAIN

Don't point that thing at me

When we define our exception for `findPath`, which will we choose as our superclass?

Reading 6, Part 2: Exceptions

Exceptions for signaling bugs

Exceptions for special results

Checked and unchecked exceptions

Throwable hierarchy

Exception design considerations

Abuse of exceptions

Next: Summary

- Throwable
- Exception
- Error
- RuntimeException

When we declare new checked exceptions, we subclass `Exception`.

CHECK

EXPLAIN

Exception design considerations

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

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

This suggests a more refined rule:

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

Here are some examples of applying this rule to hypothetical methods:

- `Queue.pop()` throws an *unchecked* `EmptyQueueException` when the queue is empty, because it's reasonable to expect the caller to avoid this with a call like `Queue.size()` or `Queue.isEmpty()`.
- `Url.getWebPage()` throws a *checked* `IIOException` when it can't retrieve the web page, because it's not easy for the caller to prevent this.

- int integerSquareRoot(int x) throws a checked NotPerfectSquareException when x has no integral square root, because testing whether x is a perfect square is just as hard as finding the actual square root, so it's not reasonable to expect the caller to prevent it.

Reading 6, Part 2:

Exceptions

Exceptions for signaling bugs

Exceptions for special results

Checked and unchecked exceptions

Throwable hierarchy

Exception design considerations

Abuse of exceptions

Next: Summary

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

Abuse of exceptions

Here's an example from *Effective Java* by Joshua Bloch (<http://library.mit.edu/item/001484188>) (Item 57 in the 2nd edition).

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

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

It is supposed to be equivalent to:

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

Or (using appropriate type T) to:

```
for (T x : a) {
    x.f();
}
```

The exception-based idiom, Bloch writes:

... is a misguided attempt to improve performance based on the faulty reasoning that, since the VM checks the bounds of array accesses, the normal loop termination test (`i < a.length`) is redundant and should be avoided.

However, because exceptions in Java are designed for use only under exceptional circumstances, few, if any, JVM implementations attempt to optimize their performance. On a typical machine, the exception-based idiom runs 70 times slower than the standard one when looping from 0 to 99.

Reading 6, Part 2: Exceptions

Exceptions for signaling bugs

Exceptions for special results

Checked and unchecked exceptions

Throwable hierarchy

Exception design considerations

Abuse of exceptions

Next: Summary

Must worse than that, the exception-based idiom is not even guaranteed to work! Suppose the computation of `f()` in the body of the loop contains a bug that results in an out-of-bounds access to some unrelated array. What happens?

If a reasonable loop idiom were used, the bug would generate an uncaught exception, resulting in immediate thread termination with a full stack trace. If the misguided exception-based loop were used, the bug-related exception would be caught and misinterpreted as a normal loop termination.

READING EXERCISES

Throw all the things!

Examine this code for doing some things:

```
static void doEverything() {
    doThingsInOrder();
}

static void doThingsInOrder() {
    try {
        for (Thing t : ALL_THE_THINGS) {
            doTheThing(t);
        }
    } catch (ThingException te) {
        return;
    }
}

static void doTheThing(Thing t) throws ThingException {
    // ...
    // ... maybe go off the end of an array
    // ...
}
```

You are not
logged in.

ThingException is an unchecked exception.

Which exceptions could be thrown by a call to `doEverything` ?

Reading 6, Part 2:

Exceptions

Exceptions for signaling bugs

Exceptions for special results

Checked and unchecked exceptions

Throwable hierarchy

Exception design considerations

Abuse of exceptions

Next: Summary

- ✖ `ArrayIndexOutOfBoundsException`
- `IOException`
- `NullPointerException`
- `ThingException`
- `OutOfMemoryError`

➤ The “maybe go off the end of an array” comment is a trick:

`ArrayIndexOutOfBoundsException`, `NullPointerException`, and `OutOfMemoryError` are all unchecked exceptions, and any of them could be thrown by `doTheThing`, propagate up through `doThingsInOrder`, and be thrown by `doEverything`.

`ThingException` is also unchecked, but there is no code in `doThingsInOrder` to throw it, and any of them thrown by `doTheThing` will be caught by the try-catch.

`IOException` is a checked exception, so these methods would be required by the compiler to declare it if it could be thrown.

CHECK

EXPLAIN

A terrible thing

What might happen if `doTheThing` throws a `ThingException` in that code?

- ✖ The program might crash
- We might fail to call `doTheThing` on all of the `Thing` objects in `ALL_THE_THINGS`
- We might call `doTheThing` multiple times on some `Thing` object(s)

➤ Notice how the try-catch is outside the for loop in the code.

If we are only partially through our iteration over `ALL_THE_THINGS`, the exception will bubble up out of the loop and prematurely end our iteration.

If we wanted to continue iterating past a `Thing` that causes a `ThingException`, we would need to put the try-catch inside the loop.

The program will not crash, because we do catch the `ThingException` before it can propagate further up the call stack.

CHECK

EXPLAIN

Collaboratively authored with contributions from:
Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama.

Next: Summary (./#summary)

Reading 6, Part 2:

Exceptions

MIT EECS accessibility (<https://accessibility.mit.edu>)

Exceptions for signaling
bugs

Exceptions for special
results

Checked and unchecked
exceptions

Throwable hierarchy

Exception design
considerations

Abuse of exceptions

Next: Summary

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/):

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

```
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;
}
```

Note that 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.

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

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 `findFirst` and `findLast`, each resolving the underdeterminedness in its own (fully deterministic) way. A client of `findOneOrMore,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.

READING EXERCISES

Distinguished

Consider this spec:

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

Which input(s) demonstrate that `findFirst` does not satisfy the spec?

- [1, 2, 2], 2
- [1, 2, 3], 2
- [1, 2, 2], 4
- none of the above, `findFirst` does satisfy the spec!

Which input(s) demonstrate that `findLast` does not satisfy the spec?

- [1, 2, 2], 2
- [1, 2, 3], 2
- [1, 2, 2], 4
- none of the above, `findLast` does satisfy the spec!

- This spec distinguishes between the two implementations by requiring -1 when `val` is not found.

CHECK

EXPLAIN

Over/under

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

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 using 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
```

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 EXERCISES

Joint declaration

Given this specification:

```
static String join(String delimiter, String[] elements)
    effects: append together the strings in elements, but at each step,
              if there are more elements left, insert delimiter
```

Rewrite the spec so it is declarative, not operational.



- effects: returns the result of adding all elements to a new StringJoiner (<https://docs.oracle.com/javase/8/docs/api/java/util/StringJoiner.html>)(delimiter)
 - effects: returns the result of looping through elements and alternately appending an element and the delimiter
 - effects: returns concatenation of elements in order, with delimiter inserted between each pair of adjacent elements
- The first and second options clearly talk about how the implementation works (operational), whereas the third is a definition (declarative) that leaves the implementation implicit.

CHECK

EXPLAIN

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 `findCanBeMissing` in the exercises.

Diagramming specifications

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

Each point in this space represents a method implementation.

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

Introduction

Deterministic vs. underdetermined specs

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 `findFirst` and `findLast` satisfy `findOneOrMore,AnyIndex`, so they are inside the region defined by that spec.

Declarative vs. operational specs

Stronger vs. weaker specs

Diagramming specifications

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.

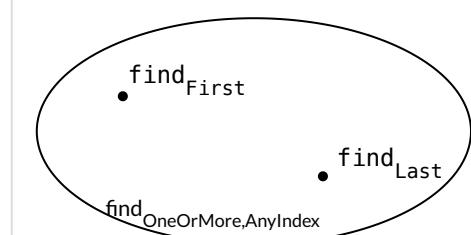
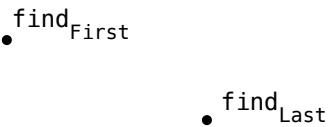
Designing good specifications

Precondition or postcondition?

About access control

About static vs. instance methods

Summary



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 `findOneOrMore,AnyIndex` by returning any index `i`, but now the spec demands the *lowest* index `i`. So there are now implementations *inside* `findOneOrMore,AnyIndex` but *outside* `findOneOrMore,FirstIndex`.

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

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

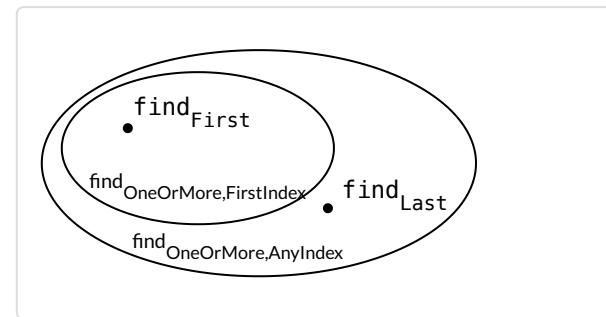
Summary

- 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 `findLast` iterates from the end of the array `arr`, it does not satisfy `findOneOrMore,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.



READING EXERCISES

Bulking up

When a specification is strengthened:

- ✖ fewer implementations satisfy it
- more implementations satisfy it
- fewer clients can use it
- more clients can use it
- none of the above

➤ A stronger spec might have a weaker precondition and/or stronger postcondition.

In both cases, the implementor must be more careful; but clients with more varied inputs or more specific needs might now be able to make use of the stronger spec.

CHECK

EXPLAIN

Strength is truth

Which of the following can be true about a pair of specifications A and B?

- ✖ 1. A can be stronger than B and have a weaker precondition
2. A can be stronger than B and have the same precondition

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

- 3. A can be stronger than B and have a stronger precondition
- 4. A can be stronger than B and have an incomparable precondition
- 5. A can be incomparable to B 

➤ If A has a stronger precondition, either it is weaker than B, or there is not a containment relationship (option 3).

If the preconditions are not comparable, then the specs will not be comparable either (4).

CHECK

EXPLAIN

Finding `findExactlyOne`

Here are the `find` specifications again:

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

```
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
```

```
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
```

```
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 already know that `findOneOrMore,FirstIndex` is stronger than `findOneOrMore,AnyIndex`, which is stronger than `findExactlyOne`.

Where is `findExactlyOne` on the diagram?

Reading 7: Designing Specifications

Introduction

Deterministic vs.
underdetermined specs

Declarative vs.
operational specs

Stronger vs. weaker
specs

Diagramming
specifications

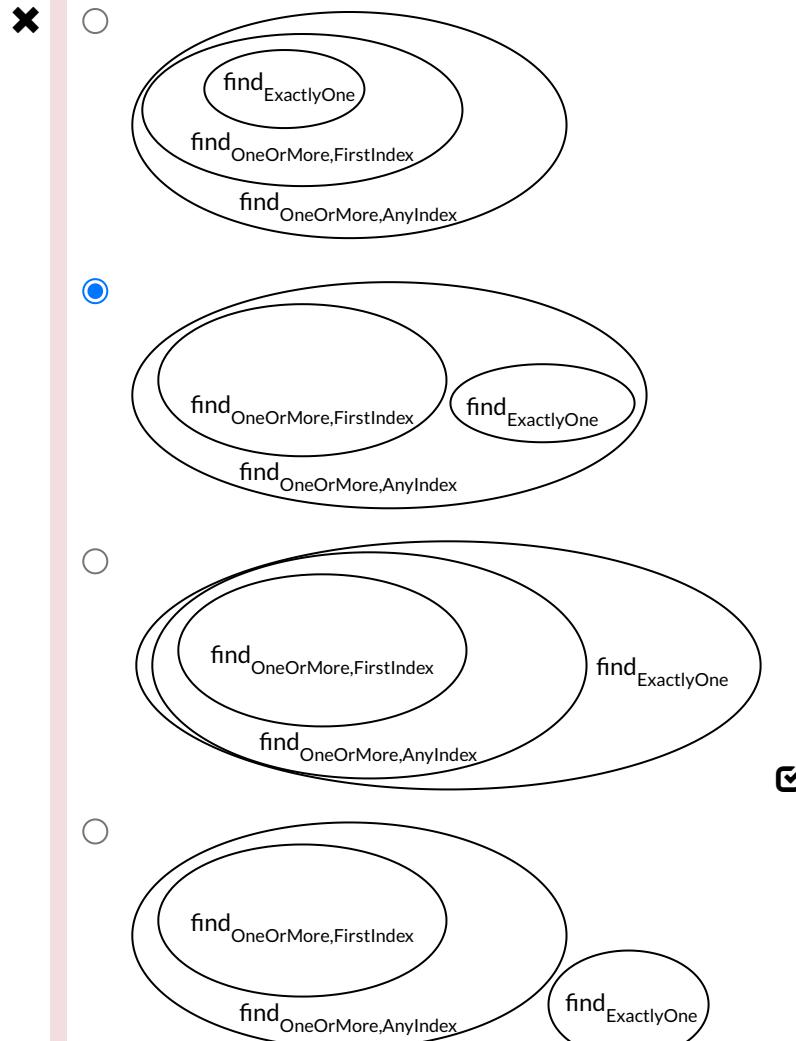
Designing good
specifications

Precondition or
postcondition?

About access control

About static vs. instance
methods

Summary



➤ `findExactlyOne` is weaker than `findOneOrMore,AnyIndex`, so its region is a superset of `findOneOrMore,AnyIndex`.

CHECK

EXPLAIN

Finding `findCanBeMissing`

Let's determine where `findCanBeMissing` is on the diagram.

How does `findCanBeMissing` compare to `findExactlyOne`?

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

- ✓ ○ `findCanBeMissing` is weaker than `findExactlyOne`
● `findCanBeMissing` is stronger than `findExactlyOne`
○ `findCanBeMissing` and `findExactlyOne` are incomparable
- `findCanBeMissing` has a weaker precondition.
For inputs that satisfy `findExactlyOne`'s precondition, `findCanBeMissing`'s postcondition is equal.
So `findCanBeMissing` is stronger.
- How does `findCanBeMissing` compare to `findOneOrMore,AnyIndex` ?**
- ✓ ○ `findCanBeMissing` is weaker than `findOneOrMore,AnyIndex`
● `findCanBeMissing` is stronger than `findOneOrMore,AnyIndex`
○ `findCanBeMissing` and `findOneOrMore,AnyIndex` are incomparable
- `findCanBeMissing` has a weaker precondition.
For inputs that satisfy `findOneOrMore,AnyIndex`'s precondition, `findCanBeMissing`'s postcondition is equal.
So `findCanBeMissing` is stronger.
- How does `findCanBeMissing` compare to `findOneOrMore,FirstIndex` ?**
- ✗ ○ `findCanBeMissing` is weaker than `findOneOrMore,FirstIndex`
○ `findCanBeMissing` is stronger than `findOneOrMore,FirstIndex`
○ `findCanBeMissing` and `findOneOrMore,FirstIndex` are incomparable
- `findCanBeMissing` has a weaker precondition.
For inputs that satisfy `findOneOrMore,FirstIndex`'s precondition, `findCanBeMissing`'s postcondition is *weaker* because it does not guarantee the lowest index i .

CHECK

EXPLAIN

Found

Where is `findCanBeMissing` on the diagram?

Reading 7: Designing Specifications

Introduction

Deterministic vs.
underdetermined specs

Declarative vs.
operational specs

Stronger vs. weaker
specs

Diagramming
specifications

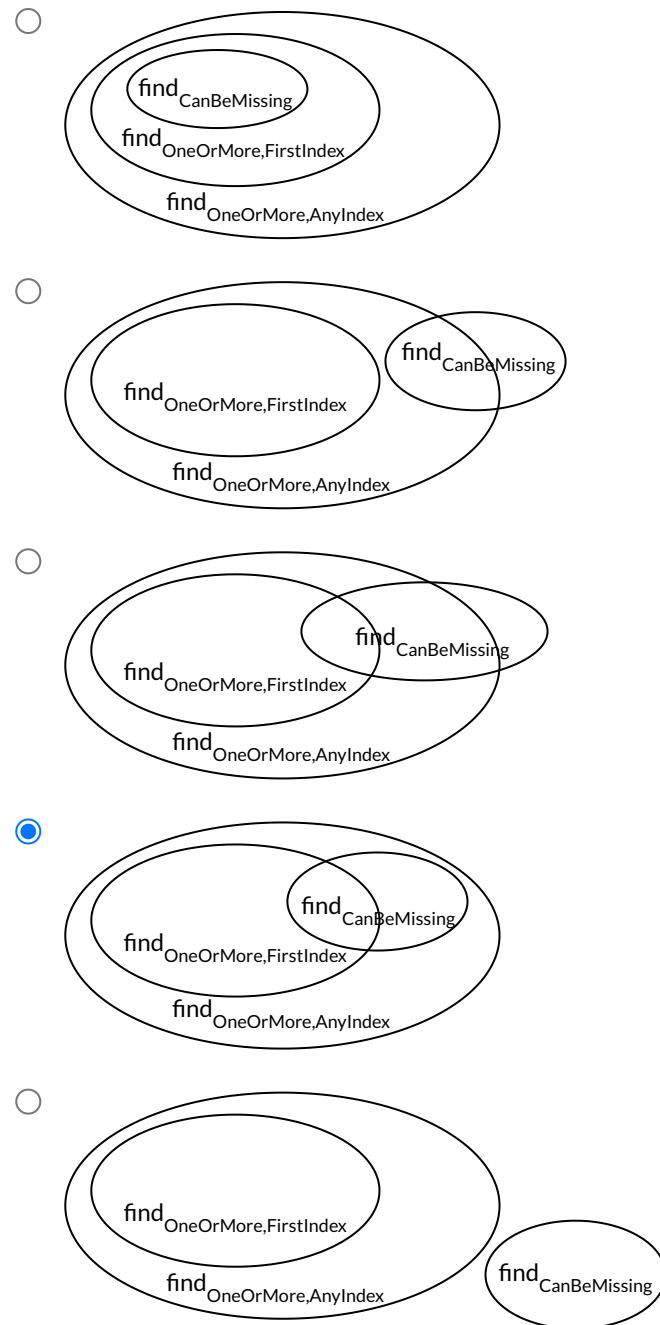
Designing good
specifications

Precondition or
postcondition?

About access control

About static vs. instance
methods

Summary



➤ Since `findCanBeMissing` is stronger than `findOneOrMore,AnyIndex`, it must be contained within that region of the space.

Then the question is its relationship to `findOneOrMore,FirstIndex`.

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

There exist implementations that satisfy `findOneOrMore,FirstIndex` but not `findCanBeMissing`: for example, they do not return -1 when `val` is not in `a`, which is excluded by `findOneOrMore,FirstIndex`'s precondition.

There also exist implementations that satisfy `findCanBeMissing` but not `findOneOrMore,FirstIndex`: for example, they do not return the lowest index when `val` occurs multiple times.

And there exist implementations that satisfy both: they can handle the weaker precondition, and the stronger parts of each postcondition.

So `findCanBeMissing` overlaps `findOneOrMore,FirstIndex`, but neither contains the other.

CHECK

EXPLAIN

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/#smelly_example_3](#)):

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

```
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 (./03-code-review/#dont_use_global_variables) and printing instead of returning (./03-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:

```
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 a checked exception for a bad argument but allowing arbitrary mutations, because a client won't be able to determine what mutations have actually been made. Here's a specification illustrating this flaw (and also written in an inappropriately operational style):

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

```
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?

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

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** (<http://docs.oracle.com/javase/tutorial/java/package/index.html>) (7 short pages) in the Java Tutorials.

Read: **Controlling Access** (<http://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**.

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

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** (<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!

READING EXERCISES

You are not

Show me a sign

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

Which of the following are signs of an excellent specification:

- 1. the specification is declarative
- 2. the specification is operational
- 3. the specification is as super-strong as possible
- 4. the specification is as super-weak as possible
- 5. the implementation is allowed to ignore invalid arguments
- 6. the implementation is allowed to use different algorithms depending on the arguments
- 7. the specification utilizes the reader's knowledge of the implementation

➤ We prefer declarative specs (option 1) to operational ones (2).

We want a spec that is neither too strong (3) nor too weak (4) to balance the constraints of implementor and client.

Although preconditions allow the implementation to ignore invalid arguments, an excellent spec would not use preconditions this way. It's better to have clear specs and implementations that fail fast than allow the implementation to quietly fail (5).

And we do not want the client to have to read the implementation at all (7).

CHECK

EXPLAIN

That's an odd way of looking at it

```
public static int secondToLastIndexOf(int[] arr, int val)
    requires: val appears in arr an odd number of times
    effects: returns the 2nd-largest i such that arr[i] == val
```

Which of the following are reasonable criticisms of this spec?

- The spec is not well-defined, we cannot implement it
- The spec is not coherent
- The spec is not deterministic
- The spec is not operational

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

➤ This is a terrible specification.

The spec is not well-defined: if `val` appears in `arr` once, what value is returned? There is no possible return value consistent with the postcondition!

The spec is not coherent: why is it limited to odd counts of `val`? Why can we only find the 2nd-largest `i`? It would be better to provide a more generally-useful procedure.

The spec is deterministic, as far as we can classify it. It is not well-defined when `val` appears in `arr` once, but this is not a case of underdeterminedness: rather than allowing *multiple* possible return values, any of which would satisfy the postcondition, the spec allows *no* possible return values.

Finally, although it's true that the spec is not operational, we actually prefer declarative specs, so this is not a reasonable criticism.

CHECK

EXPLAIN

Behavioral oddities

```
public static int secondToLastIndexOf(int[] arr, int val)
    requires: val appears in arr an odd number of times
    effects: returns the 2nd-largest i such that arr[i] == val
```

Consider the following test cases for `secondToLastIndexOf`:

[1, 3, 4], 3 returns 1

✗ valid test case

could be valid with a weaker precondition, same postcondition

could be valid with a weaker precondition, stronger postcondition

could be valid with same precondition, weaker postcondition

➤ The current postcondition doesn't admit any output value with only one occurrence of `val`.

[1, 3, 3, 4], 3 returns 1

✗ valid test case

could be valid with a weaker precondition, same postcondition

could be valid with a weaker precondition, stronger postcondition

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

could be valid with same precondition, stronger postcondition

➤ We would need to weaken the precondition to allow even occurrences of `val`.

[`1, 3, 3, 3, 4`], `3` returns `2`

✗ valid test case

could be valid with a weaker precondition, same postcondition

could be valid with a weaker precondition, stronger postcondition

could be valid with same precondition, stronger postcondition

➤ Satisfies the precondition, and the postcondition is deterministic.

[`3, 3, 3, 3`], `3` throws an exception

✗ valid test case

could be valid with a weaker precondition, same postcondition

could be valid with a weaker precondition, different postcondition

could be valid with same precondition, stronger postcondition

➤ This is not (currently) a valid test because it violates the precondition; maybe the method throws an exception in such cases, but that behavior is beyond the spec, so the test can't depend on it. So at least we need to weaken the precondition to allow this input. Having done that, we also need to change the postcondition to specify that an exception is thrown when `val` appears an odd number of times, because the current postcondition doesn't mention any exceptions at all.

CHECK

EXPLAIN

Odd doc

```
public static int secondToLastIndexOf(int[] arr, int val)
requires: val appears in arr an odd number of times
effects: returns the 2nd-largest i such that arr[i] == val
```

Choose lines below to complete one possible Javadoc version of this terrible spec:

✗ /*

/**

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

- * Finds the second-to-last occurrence of a value in an array.
- * Find j, the largest index such that arr[j] == val.
- * Then find i, the largest index such that i < j and arr[i] == val.
- * @param arr array to search
- * @param arr fixed-size array of integers to search
- * @param val value to search for
- * @param val value to search for, requires val appears in arr an odd number of times
-
- * @return index i
- * @return second-largest index i such that arr[i] == val
- */

➤ The business about computing j and then i in terms of j is operational, we don't need that. We don't need to explain Java semantics (e.g. arr is a fixed-size array of integers), but we definitely do need to explain the precondition. It's good to include a summary of the function as the first line of the Javadoc, but we should still explain the postcondition on the return value clearly and succinctly.

And let's just be clear: regardless of how well we write it up, *this is a terrible specification*.

CHECK

EXPLAIN

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:

Reading 7: Designing Specifications

Collaboratively authored
Introduction with contributions from:
Saman Amarasinghe, Adam
~~Deterministic vs.~~
~~Underdetermined specs~~,
Michael Ernst, Max
Goldman, John Gutttag,
Daniel Jackson, Rob Miller,
Martin Rihard, and
~~Declarative vs.~~
~~Operational specs~~.
This work is licensed under
Stronger vs. weaker
(<http://creativecommons.org/licenses/by-sa/4.0/>).

Diagramming specifications

Designing good specifications

Precondition or postcondition?

About access control

About static vs. instance methods

Summary

- **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.

MIT EECS accessibility (<https://accessibility.mit.edu>)

Reading 08: Avoiding
Debugging

First Defense: Make
Bugs Impossible

Second Defense:
Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental
Development

Modularity &
Encapsulation

Reading 08: 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** (http://web.mit.edu/6.005/www/fa15/classes/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 (http://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.

READING EXERCISES

Final references, immutable objects

Consider the following code, executed in order:

```
char vowel0 = 'a';  
final char vowel1 = vowel0;  
  
String vowel2 = vowel1 + "eiou";  
final String vowel3 = vowel2;  
  
char[] vowel4 = new char[] { vowel0, 'e', 'i', 'o', 'u' };  
final char[] vowel5 = vowel4;
```

Which of the following statements are legal Java (i.e. produce no compiler error if placed after the code above)?

- vowel0 = 'y';
- vowel1 = vowel0;
- vowel2 = "uoie" + vowel1;
- vowel3 = vowel2;
- vowel2[0] = 'x';
- vowel3[0] = 'x';
- vowel4 = vowel5;

Reading 08: Avoiding Debugging

First Defense: Make Bugs Impossible

Second Defense: Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental Development

Modularity & Encapsulation

Reading 08: Avoiding Debugging

First Defense: Make Bugs Impossible

Second Defense: Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental Development

Modularity & Encapsulation

- vowel5 = vowel4;
- vowel4[0] = 'x'; 
- vowel5[0] = 'z'; 

➤ vowel1, vowel3, and vowel5 are final, so they cannot be reassigned.

vowel2 and vowel3 are Strings, which don't provide the [] indexing operation like arrays do. You have to use the charAt() method instead.

Even though vowel5 is final, it points at a mutable array, so vowel5[0] can be reassigned.

CHECK

EXPLAIN

Afterwards

After all the legal statements in the previous exercise's answers are executed, what is the resulting value of each variable? Write just the sequence of letters found in the variable's value, with no punctuation or spaces. For example:

vowel0

y

➤ vowel0 was reassigned to 'y' in the exercise answers.

vowel1

 dfg

a

➤ vowel1 is final and refers to an immutable type (char), so it must always be the value it was initialized with, 'a'.

vowel2

 v

uoiea

➤ vowel2 was reassigned to "uoiea" in the exercise answers.

vowel3

 cvb

aeiou

Reading 08: Avoiding Debugging

First Defense: Make Bugs Impossible

Second Defense: Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental Development

Modularity & Encapsulation

➤ vowel3 is final and refers to an immutable type (`String`), so it must always be the value it was initialized with, "aeiou".

vowel4

✗ fgh

zeiou

➤ vowel4 and vowel5 both refer to the same array object, which is initially has the five elements `a`, `e`, `i`, `o`, `u`. Mutating that object through either `vowel4[0]` or through `vowel5[0]` will affect what both variables see. The final mutation sets the 0th element to '`z`'.

vowel5

✗ cvb

zeiou

➤ vowel4 and vowel5 refer to the same array object, so the answer for vowel5 is the same as for vowel4.

CHECK

EXPLAIN

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

Reading 08: Avoiding Debugging

First Defense: Make Bugs Impossible

Second Defense: Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental Development

Modularity & Encapsulation

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 (<http://web.mit.edu/6.005/www/fa15/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.

Reading 08: Avoiding Debugging

First Defense: Make Bugs Impossible

Second Defense: Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental Development

Modularity & Encapsulation

What to Assert

Reading 08: Avoiding
Debugging

First Defense: Make
Bugs Impossible

Second Defense:
Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental
Development

Modularity &
Encapsulation

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.fail();  
}
```

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;
```

Reading 08: Avoiding Debugging

First Defense: Make Bugs Impossible

Second Defense: Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental Development

Modularity & Encapsulation

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;
```

Reading 08: Avoiding Debugging

First Defense: Make Bugs Impossible

Second Defense: Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental Development

Modularity & Encapsulation

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.

READING EXERCISES

Assertions

Consider this (incomplete) function:

```
/**
 * Solves quadratic equation ax^2 + bx + c = 0.
 *
 * @param a quadratic coefficient, requires a != 0
 * @param b linear coefficient
 * @param c constant term
 * @return a list of the real roots of the equation
 */
public static List<Double> quadraticRoots(final int a, final int b, final int
c) {
    List<Double> roots = new ArrayList<Double>();
    // A
    ... // compute roots
    // B
    return roots;
}
```

What statements would be reasonable to write at position A?

- assert a != 0;
- assert b != 0;
- assert c != 0;
- assert roots.size() >= 0;
- assert roots.size() <= 2;
- for (double x : roots) { assert Math.abs(a*x*x + b*x + c) < 0.0001; }

Reading 08: Avoiding Debugging

First Defense: Make Bugs Impossible

Second Defense: Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental Development

Modularity & Encapsulation

What statements would be reasonable to write at position B?

- ✗ assert a != 0;
 assert b != 0;
 assert c != 0;
 assert roots.size() >= 0;
 assert roots.size() <= 2; ↗
 for (double x : roots) { assert Math.abs(a*x*x + b*x + c) < 0.0001; } ↗

➤ It's good to assert `a!=0` at the start of the function, but not necessary to assert at the end of the function, since `final` prevents it from being changed.

It isn't correct to assert `b!=0` or `c!=0`, since 0 is a legal value for those parameters.

It isn't reasonable to assert the content of the `roots` list until it has actually been computed – i.e., not at the start of the function, but at the end of the function.

A quadratic equation can have at most 2 roots, so it's reasonable to assert `roots.size()<=2` at the end of the function.

It isn't reasonable to assert `roots.size()>=0`, because lists are guaranteed to have nonnegative length. This assertion would only fail if the List `size()` method were broken. Don't assert guaranteed behavior of the language or library.

Finally, the `for` loop checks that each of returned roots is actually a root, causing the quadratic polynomial to evaluate to 0 (or close enough, within some margin for error to account for the imprecision of floating-point numbers). This is a very reasonable assertion of the postcondition of the method.

CHECK

EXPLAIN

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 (http://web.mit.edu/6.005/www/fa15/classes/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 (http://web.mit.edu/6.005/www/fa15/classes/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.

Reading 08: Avoiding Debugging

First Defense: Make Bugs Impossible

Second Defense: Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental Development

Modularity & Encapsulation

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 (<http://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:

Reading 08: Avoiding Debugging

First Defense: Make Bugs Impossible

Second Defense: Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental Development

Modularity & Encapsulation

```
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.

Reading 08: Avoiding Debugging

First Defense: Make Bugs Impossible

Second Defense: Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental Development

Modularity & Encapsulation

- **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.

READING EXERCISES

Variable scope

Consider the following code (which is missing some variable declarations):

```
1 class Apartment {  
2     Apartment(String newAddress) {  
3         this.address = newAddress;  
4         this.roommates = new HashSet<Person>();  
5     }  
6  
7     String getAddress() {  
8         return address;  
9     }  
10    void addRoommate(Person newRoommate) {  
11        roommates.add(newRoommate);  
12        if (roommates.size() > MAXIMUM_OCCUPANCY) {  
13            roommates.remove(newRoommate);  
14            throw new TooManyPeopleException();  
15        }  
16    }  
17  
18    void getMaximumOccupancy() {  
19        return MAXIMUM_OCCUPANCY;  
20    }  
21 }
```

You are not logged in.

Which of these lines are within the scope of the newRoommate variable?

- ✗ line 3
 line 8
 line 12
 line 15
 line 20

➤ The scope of a parameter is its function body, so lines 10-16 are in scope.

What would be the scope for the (currently undeclared) address variable?

- ✓ lines 2-21
 lines 3-4
 line 8
 lines 12-16

➤ `address` is an instance variable, also called a field in Java, so its scope should be the entire class. If you're unclear about instance variables, reread Classes and Objects (http://web.mit.edu/6.005/www/fa15/classes/02-basic-java/#classes_and_objects) and its associated Java tutorial paths, or ask in office hours.

Out of the choices below, what is the best declaration for the roommates variable?

- ✓ `List<Person> roommates;`
 `Set<Person> roommates;`
 `final Set<Person> roommates;`
 `HashSet<Person> roommates;`

➤ The `roommates` variable is a `Set`, not a `List`, and it's best to use abstract types like `Set` rather than concrete types like `HashSet` when declaring a variable. Include `final` because the `roommates` variable should always point to the same `Set` object. It never needs to be reassigned, only mutated.

Out of the choices below, what is the best declaration for the MAXIMUM_OCCUPANCY variable?

- ✗ `int MAXIMUM_OCCUPANCY = 8;`
 `final int MAXIMUM_OCCUPANCY = 8;`
 `static int MAXIMUM_OCCUPANCY = 8;`

Reading 08: Avoiding Debugging

First Defense: Make Bugs Impossible

Second Defense: Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental Development

Modularity & Encapsulation

○ static final int MAXIMUM_OCCUPANCY = 8; 

- MAXIMUM_OCCUPANCY is a constant, so it should be declared `static` (not an instance variable, but a class variable) and `final` (so that it can never be reassigned).

CHECK

EXPLAIN

Reading 08: Avoiding Debugging

First Defense: Make Bugs Impossible

Second Defense: Localize Bugs

Assertions

What to Assert

What Not to Assert

Incremental Development

Collaboratively authored
 from:

Encapsulation
Saman Amarasinghe, Adam Chlipala, Srinivas Devadas, Michael Ernst, Max Goldman, John Gutttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama.
This work is licensed under CC BY-SA 4.0
(<http://creativecommons.org/licenses/by-sa/4.0/>).

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.

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

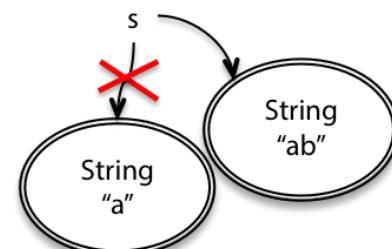
Mutability

Recall from *Basic Java* when we discussed snapshot diagrams (http://web.mit.edu/6.005/www/fa15/classes/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` (<http://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` (<http://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:

Reading 9: Mutability & Immutability

Mutability

Risks of mutation

Aliasing is what makes mutable types risky

Specifications for mutating methods

Iterating over arrays and lists

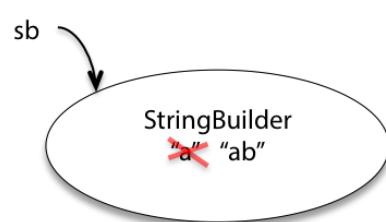
Mutation undermines an iterator

Mutation and contracts

Useful immutable types

Summary

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

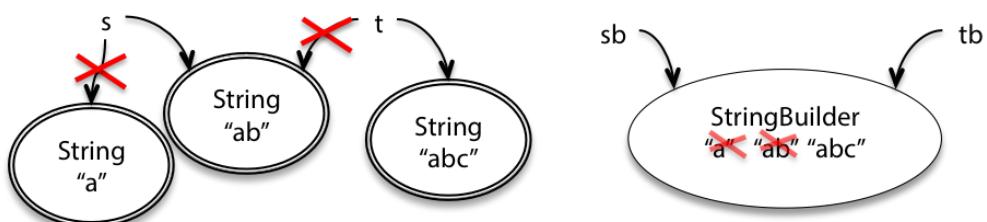


`StringBuilder` has other methods as well, for deleting parts of the string, inserting in the middle, or changing individual characters.

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, like this:

```
String s = "";
for (int i = 0; i < n; ++i) {
    s = s + n;
}
```

Using immutable strings, this makes a lot of temporary copies — the first number of the string ("0") is actually copied n times in the course of building up the final string, the second number is copied $n-1$ times, and so on. It actually costs $O(n^2)$ time just to do all that copying, even though we only concatenated n elements.

`StringBuilder` is designed to minimize this copying. It uses a simple but clever internal data structure to avoid doing any copying at all until the very end, when you ask for the final `String` with a `toString()` call:

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

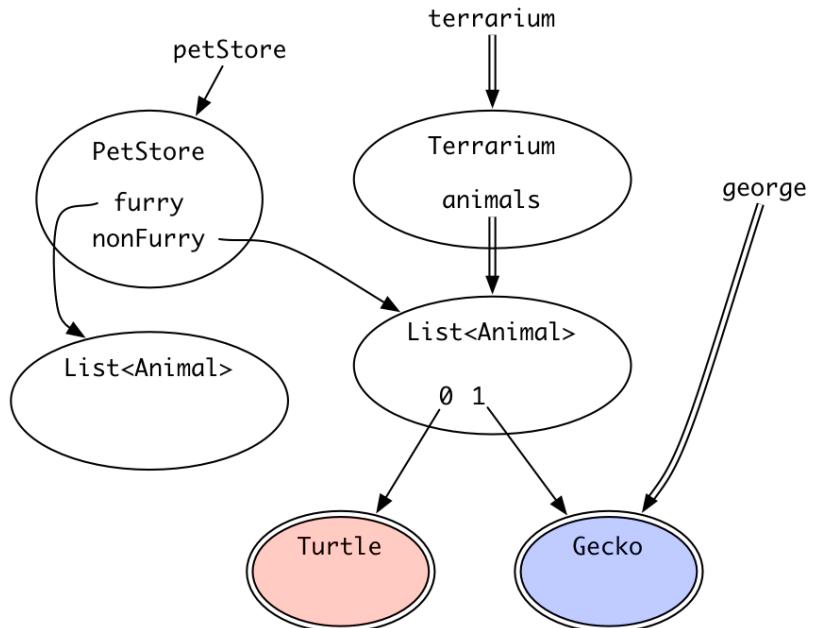
Summary

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

Getting good performance is one reason why we use mutable objects. Another is convenient sharing: two parts of your program can communicate more conveniently by sharing a common mutable data structure.

READING EXERCISES

Follow me



Can a client with the variable `terrarium` modify the `Turtle` in red?

- ✖ No, because the `terrarium` reference is immutable
- ✖ No, because the `Turtle` object is immutable ➔
- ✖ Yes, because the reference from list index 0 to the `Turtle` is mutable
- ✖ Yes, because the `Turtle` object is mutable
- The double circle means it's immutable. Whether it can be reached via mutable or immutable references is irrelevant.

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

Can a client with the variable `george` modify the Gecko in blue?

- ✖ No, because the `george` reference is immutable
 No, because the `Gecko` object is immutable ✅
 Yes, because the reference from list index 1 to the `Gecko` is mutable
 Yes, because the `Gecko` object is mutable

➤ The double circle means it's immutable. Whether it can be reached via mutable or immutable references is irrelevant.

Can a client with just the variable `petStore` make it impossible for a client with just the variable `terrarium` to reach the Gecko in blue?

Choose the best answer.

- ✖ No, because the `terrarium` reference is immutable
 No, because the `Gecko` object is immutable
 Yes, because the `petStore` reference is mutable
 Yes, because the `PetStore` object is mutable
 Yes, because the `List` object is mutable ✅
 Yes, because the reference from list index 1 to the `Gecko` is mutable

➤ The only mutable reference between `terrarium` and the blue `Gecko` is the reference from list index 1. If we want to make the `Gecko` unreachable from `terrarium`, ultimately that's the reference we have to break.

However, that mutable reference is inside a `List` object. We do not have direct access to it. And if `List` was immutable, part of its job would be to prevent that mutable reference from being reassigned.

But `List` is mutable, and it provides methods like `set` or `remove` that will allow us to change that internal mutable reference. That's the best answer to this question.

CHECK

EXPLAIN

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

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

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 (http://en.wikipedia.org/wiki/Don't_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:

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

```
// 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?

READING EXERCISES

Risky #1

What will the code print?

 , 10

 么 10

 Since `sumAbsolute` mutates the list, the call to `sum` is now summing a list of positive values.

CHECK

EXPLAIN

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?

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

Let's consider Date (<http://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:

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

```
// 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?)

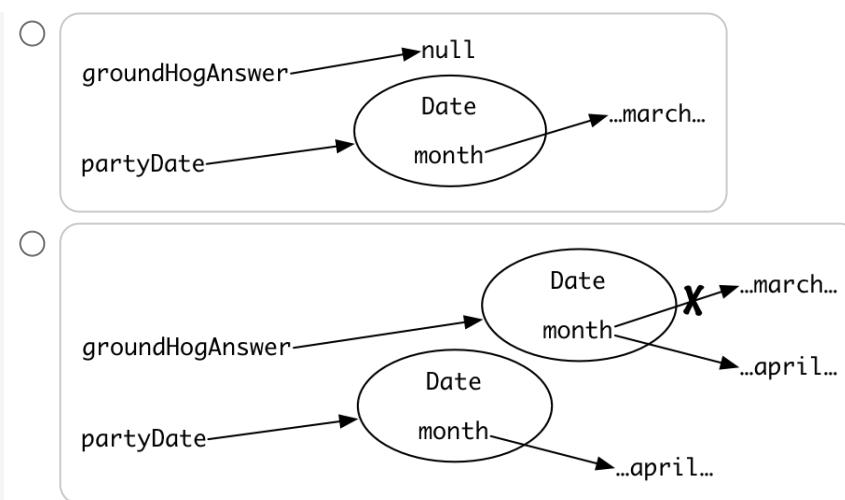
What happens when these two decisions interact? Even worse, think about who will first discover this bug — will it be `startOfSpring()`? Will it be `partyPlanning()`? Or will it be some completely innocent third piece of code that also calls `startOfSpring()`?

READING EXERCISES

Risky #2

We don't know how `Date` stores the month, so we'll represent that with the abstract values `...march...` and `...april...` in an imagined `month` field of `Date`.

Which of these snapshot diagrams shows the bug?



Reading 9: Mutability & Immutability

Mutability

Risks of mutation

Aliasing is what makes mutable types risky

Specifications for mutating methods

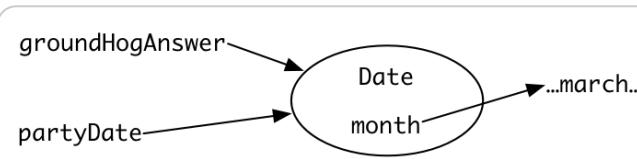
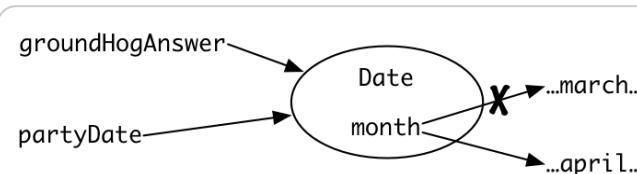
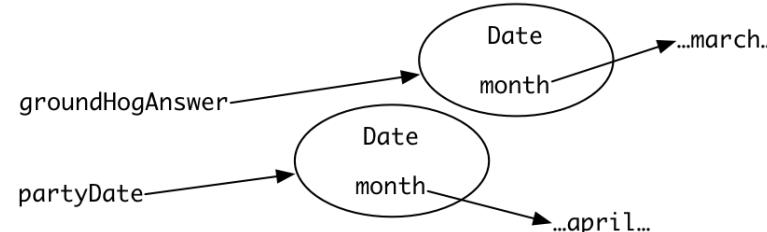
Iterating over arrays and lists

Mutation undermines an iterator

Mutation and contracts

Useful immutable types

Summary



CHECK

Understanding risky example #2

A second bug

The code has another potential bug in how it adds to the month.

Take a look at the Java API documentation for `Date.setMonth`.

For what result of `partyDate.getMonth()` could there be a problem?

2 11

➤ According to the specs of `getMonth` and `setMonth`, 0 represents January and 11 represents December.

So if `partyDate` is in December, `getMonth` returns 11, and the code makes an invalid call to `setMonth(12)`.

CHECK

EXPLAIN

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

NoSuchMonthException

The documentation for `Date.setMonth` says `month`: the month value between 0–11.

Based on that statement and just what you've read so far...

Which of the following *might* happen when this month-adding bug is triggered?

- The method call will do nothing
- The method call will actually do the thing we wanted
- The method call will cause the `Date` object to become invalid and report incorrect values
- The method call will throw a checked exception
- The method call will throw an unchecked exception
- The method call will set our computer clock to 9/9/99
- The method call will cause other `Date` objects to become invalid
- The method call will never return

CHECK

SuchTerribleSpecificationsException

Elsewhere in the documentation for `Date`, it says: “arguments given to methods [...] need not fall within the indicated ranges; for example, a date may be specified as January 32 and is interpreted as meaning February 1”.

What looks like a precondition... isn't!

Which of these is an argument against this feature of `Date`?

- ✗ Don't repeat yourself (DRY)
- Fail fast ✓
- Groundhog algorithm
- Exceptions for special results
- Preconditions restrict the client

Reading 9: Mutability & Immutability

Mutability

Risks of mutation

Key points:

Aliasing is what makes mutable types risky

- **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.

Iterating over arrays and lists

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.

Mutation undermines an iterator

In fact, you should never use `Date`! Use one of the classes from package `java.time`

(<http://docs.oracle.com/javase/8/docs/api/index.html?java/time/package-summary.html>): `LocalDateTime`

(<http://docs.oracle.com/javase/8/docs/api/?java/time/LocalDateTime.html>), `Instant`

(<http://docs.oracle.com/javase/8/docs/api/?java/time/Instant.html>), etc. All guarantee in their specifications that they are *immutable*.

Summary

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

CHECK

EXPLAIN

- Some clients might appreciate the flexibility of this approach, but other clients will have bugs that `Date` makes it harder to find. If the spec says values need to be in particular ranges, it would be better to fail fast when they're not.

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

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 (http://web.mit.edu/6.005/www/fa15/classes/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

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

The next mutable object we're going to look at is an **iterator** — an object that steps through a collection of elements and returns the elements one by one. Iterators are used under the covers in Java when you're using a `for` loop to step through a `List` or array. This code:

```
List<String> 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` (<http://docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html>).

Before we go any further:

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

Read: **the final keyword** (<http://www.codeguru.com/java/tij/tij0071.shtml>) on CodeGuru.

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

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

`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 (<http://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** (<http://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** (<http://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 (<http://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 (<http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>)).

`final` is used to indicate which parts of the object's internal state can change 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).

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

```
/**  
 * 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;
```

```
}
```

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

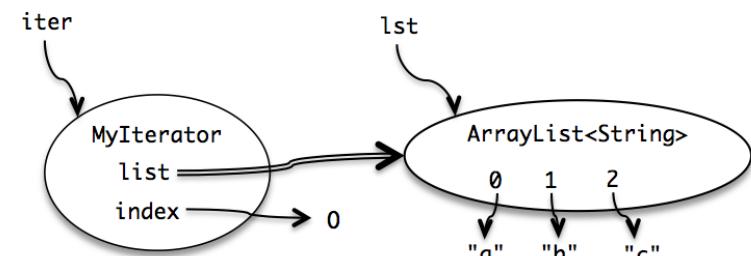
Useful immutable types

Summary

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.



READING EXERCISES

MyIterator.next signature

This example is one of the first we've seen that uses *instance methods*. Instance methods operate on an instance of a class, take an implicit `this` parameter (like the explicit `self` parameter in Python), and can access *instance fields*.

Let's examine `MyIterator`'s `next` method:

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

```
public class MyIterator {  
  
    private final ArrayList<String> list;  
    private int index;  
  
    ...  
  
    /**  
     * 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;  
    }  
}
```

Thinking about `next` as an **operation** as defined in *Static Checking: Types* (<http://web.mit.edu/6.005/www/fa15/classes/01-static-checking/#types>)...

What are the types of the input(s) to `next` ?

- ✖ void – there are no inputs
- ArrayList
- MyIterator ✅
- String
- boolean
- int

➤ Because `next` is an instance method, the `MyIterator` object on which it was called (`this`) is an input. We can see from the method signature that it takes no other inputs.

What are the types of the output(s) from `next` ?

- ✖ void – there are no outputs
- ArrayList
- MyIterator

- String
- boolean
- int

➤ We can see from the method signature that `next` returns a `String`.

CHECK

EXPLAIN

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

MyIterator.next precondition

`next` has the precondition requires: `hasNext()` returns `true`.

Which input(s) to `next` are constrained by the precondition?

- ✗ none of them
 this
 `hasNext`
 element

When the precondition isn't satisfied, the implementation is free to do anything.

What does this particular implementation do when the precondition is not satisfied?

- ✗ return `null`
 return some other element of the list
 throw a checked exception
 throw an unchecked exception

CHECK

EXPLAIN

MyIterator.next postcondition

Part of the postcondition of `next` is: @return next element of the list.

Which output(s) from `next` are constrained by that postcondition?

- ✗ none of them
 this

- hasNext
- the return value

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

➤ This postcondition is on the String returned from next .

Another part of the postcondition of next is modifies: this iterator to advance it to the element following the returned element.

What is (are) constrained by that postcondition?

- nothing
- this
- hasNext
- the return value

➤ This postcondition is on the state of the MyIterator instance after we call next .

CHECK

EXPLAIN

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:

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

```
// 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.

READING EXERCISES

Draw a snapshot diagram

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:

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

```
for (String subject : subjects) {  
    if (subject.startsWith("6.")) {  
        subjects.remove(subject);  
    }  
}
```

then you'll get a `ConcurrentModificationException` (<http://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(subject);  
    }  
}
```

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!

READING EXERCISES

Pick a snapshot diagram

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.

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

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 (<http://docs.oracle.com/javase/8/docs/api/?java/util/Iterator.html>)? List (<http://docs.oracle.com/javase/8/docs/api/?java/util/List.html>)? Collection (<http://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 NoSuchElementException if nobody with username is in MIT's database  
 */  
public static char[] getMitId(String username) throws NoSuchElementException {  
    // ... 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:

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

```
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.
```

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

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[] getMitId(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 (<http://docs.oracle.com/javase/8/docs/api/?java/math/BigInteger.html>) and BigDecimal (<http://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 (<http://docs.oracle.com/javase/8/docs/api/index.html?java/time/package-summary.html>) based on the granularity of timekeeping you need.

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

- The usual implementations of Java's collections types — `List`, `Set`, `Map` — are all mutable: `ArrayList`, `HashMap`, etc. The `Collections` (<http://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` (<http://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*!

READING EXERCISES

Immutability

You are not logged in.

Which of the following are correct?

- ✗ 1. A class is immutable if all of its fields are `final`
 2. A class is immutable if instances of it always represent the same value
 3. Instances of an immutable class can be safely shared
 4. Immutability can be implemented using defensive copying
 5. Immutability allows us to reason about global properties instead of local ones

➤ Option 1 is false because final fields may still point to *mutable* objects, so merely having final references is insufficient.

Option 2 is the definition of immutable objects; (3) is one of the major benefits of immutability.

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: This reading was authored with contributions from: Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama.

This work is licensed under CC BY-SA 4.0 (<http://creativecommons.org/licenses/by-sa/4.0/>).

Defensive copying is a strategy for preventing sharing of (mutable) objects, not for making them immutable (4).

The need for global reasoning is a negative consequence of *mutability*, because contracts expand to cover more parts of the program over more time. Immutability allows us to reason locally, instead of globally (5).

CHECK

EXPLAIN

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
$n! = n \times (n - 1) \times \cdots \times 2 \times 1 = \prod_{k=1}^n k$ <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);
}

```

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

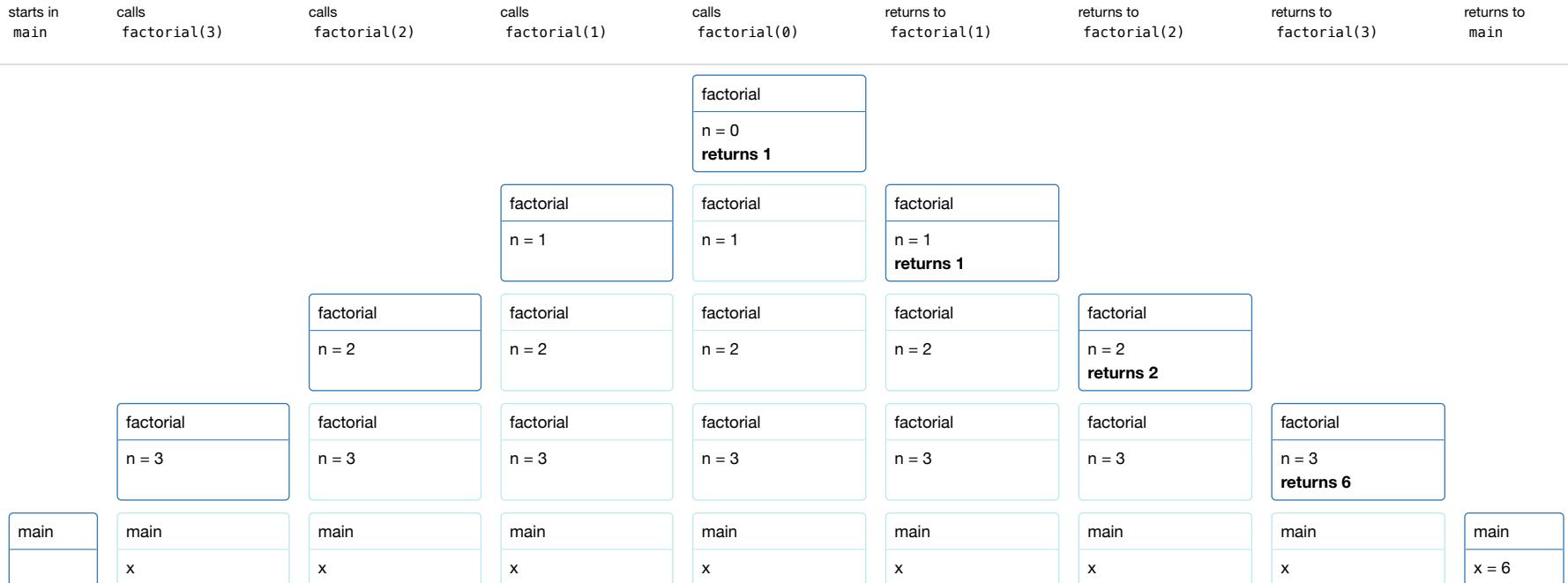
Reentrant Code

When to Use Recursion Rather Than Iteration

Common Mistakes in Recursive Implementations

Summary

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**

([http://www.pythontutor.com/visualize.html#code=public+class+Factorial+%7B%0A++++public+static+void+main\(String%5B%5D+args\)+%7B%0A++++++long+x+%3D+factorial\(3\)%3B%0A++++%7D%0A++++public+\(n+%3D%3D+0\)+%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&curInstr=0](http://www.pythontutor.com/visualize.html#code=public+class+Factorial+%7B%0A++++public+static+void+main(String%5B%5D+args)+%7B%0A++++++long+x+%3D+factorial(3)%3B%0A++++%7D%0A++++public+(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&curInstr=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**

([http://www.pythontutor.com/visualize.html#code=public+class+Fibonacci+%7B%0A++++public+static+void+main\(String%5B%5D+args\)+%7B%0A++++++int+x+%3D+fibonacci\(3\)%3B%0A++++%7D%0A++++public+\(n+%3D%3D+0+%7C%7C+n+%3D%3D+1\)+%7B%0A++++++return+1%3B%0A++++++%7D+else+%7B%0A++++++return+fibonacci\(n-1\)%3B%0A++++++%7D%0A++++%7D%0A%7D&mode=display&origin=opt-frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curInstr=0](http://www.pythontutor.com/visualize.html#code=public+class+Fibonacci+%7B%0A++++public+static+void+main(String%5B%5D+args)+%7B%0A++++++int+x+%3D+fibonacci(3)%3B%0A++++%7D%0A++++public+(n+%3D%3D+0+%7C%7C+n+%3D%3D+1)+%7B%0A++++++return+1%3B%0A++++++%7D+else+%7B%0A++++++return+fibonacci(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&curInstr=0))

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.

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 EXERCISES

Recursive factorial

Consider this recursive implementation of the factorial function.

```
public static int factorial(int n) {  
    if (n == 0) {  
        return 1; // this is called the base case  
    } else {  
        return n * factorial(n-1); // this is the recursive step  
    }  
}
```

For `factorial(3)`, how many times will the base case `return 1` be executed?

- 0 times
- 1 time
- 2 times
- 3 times
- more than 3 times

➤ `factorial(3)` calls `factorial(2)`, which calls `factorial(1)`, which calls `factorial(0)`, which executes the base case once. Each of the recursive calls then returns, without any further recursion.

CHECK

EXPLAIN

Recursive Fibonacci

Consider this recursive implementation of the Fibonacci sequence.

```
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  
    }  
}
```

For `fibonacci(3)`, how many times will the base case `return 1` be executed?

- 0 times
- 1 time
- 2 times
- 3 times
- more than 3 times

➤ `fibonacci(3)` calls `fibonacci(2)` which calls `fibonacci(1)`, which executes the base case once, and `fibonacci(0)` which executes the base case again. `fibonacci(2)` then returns to `fibonacci(3)`, which now calls `fibonacci(1)`, which executes the base case one more time, for a total of 3.

Another way to do it: because fibonacci is a pure function, with no side-effects, we can substitute expressions:

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

```
fibonacci(3) = fibonacci(2) + fibonacci(1)
              = (fibonacci(1) + fibonacci(0)) + fibonacci(1)
              reduces to (1 + 1) + 1 = 3 base cases
```

Still another way to check, which only works for Fibonacci, is to realize that each execution of the base case contributes 1 to the final answer, and none of the recursive steps contribute any additional value. So the number of base case executions in `fibonacci(n)` is the same as the nth Fibonacci number.

CHECK

EXPLAIN

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 }
```

READING EXERCISES

`subsequences("c")`

What does `subsequences("c")` return?

- "c"
- ""
- ",c"
- "c,"

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

► subsequences("c") first calls subsequences("") , which just returns "" .

We then split this empty string on "," , which returns an array of one element, the empty string.

The for loop then iterates over this array and constructs the two ways that subsequences of "c" can be formed (with or without the letter 'c'). It appends each new subsequence to result, starting it with a comma. This means we'll end up with an extra comma at the beginning of result, which we have to remove after the for loop.

- result = "" (line 10)
- result = "," (line 12)
- result = ",,c" (line 13)
- result = ",c" (line 15)

Finally subsequences("c") returns ",c" representing the empty string and the string "c" , the two possible subsequences of the one-character string "c" .

CHECK

EXPLAIN

subsequences("gc")

What does subsequences("gc") return?

- ✗ "g,c"
- ",g,c,gc"
- ",gc,g,c"
- "g,c,gc"

► subsequences("gc") first calls subsequences("c") , which returns ",c" as we saw in the previous question.

We then split this string on "," , which produces an array of two elements, "" and "c" .

The for loop then iterates over this array, producing two new subsequences from each element:

- result = "" (line 10)
- result = "," (line 12)
- result = ",,g" (line 13)
- result = ",,g,c" (line 12)
- result = ",,g,c,gc" (line 13)
- result = ",,g,c,gc" (line 15)

This final result is returned from subsequences("gc") .

CHECK

EXPLAIN

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.

READING EXERCISES

Recursive structure

Recursive methods have a base case and a recursive step. What other concepts from computer science also have (the equivalent of) a base case and a recursive step?

- proof by induction
- regression testing
- recursive functions
- binary trees

► In proof by induction, the base case is called a base case, and the recursive step is often called the inductive step.

Binary trees have leaves (base case) and internal nodes (recursive step).

CHECK

EXPLAIN

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

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". At first, we would select "o" to be in the partial subsequence, and recursively extend it with all subsequences of "range". Then, we would 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.

READING EXERCISES

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

Unhelpful 1

Louis Reasoner doesn't want to use a helper method, so he tries to implement `subsequences()` by storing `partialSubsequence` as a static variable instead of a parameter. Here is his implementation:

```
private static String partialSubsequence = "";
public static String subsequencesLouis(String word) {
    if (word.isEmpty()) {
        // base case
        return partialSubsequence;
    } else {
        // recursive step
        String withoutFirstLetter = subsequencesLouis(word.substring(1));
        partialSubsequence += word.charAt(0);
        String withFirstLetter = subsequencesLouis(word.substring(1));
        return withoutFirstLetter + "," + withFirstLetter;
    }
}
```

Suppose we call `subsequencesLouis("c")` followed by `subsequencesLouis("a")`.

What does `subsequencesLouis("c")` return?

- ✗ "c"
✗ ""
✗ ",c"
✗ "c,"

What does `subsequencesLouis("a")` return?

- ✗ "a"
✗ ""
✗ ",a"
✗ "a,"
✗ "c,ca"

➤ The static variable maintains its value across calls to `subsequencesLouis()`, so it still has the final value "c" from the call to `subsequencesLouis("c")` when `subsequencesLouis("a")` starts. As a result, every subsequence of that second call will have an extra c before it.

CHECK

EXPLAIN

Unhelpful 2

Louis fixes that problem by making `partialSubsequence` public:

```
/** 
 * Requires: caller must set partialSubsequence to "" before calling subsequencesLouis().
 */
public static String partialSubsequence;
```

Alyssa P. Hacker throws up her hands when she sees what Louis did. Which of these statements are true about his code?

- ✗ `partialSubsequence` is risky – it should be final
✗ `partialSubsequence` is risky – it is a global variable
✗ `partialSubsequence` is risky – it points to a mutable object

► `partialSubsequence` is indeed a global variable. It can't be made final, however, because the recursion needs to reassign it (frequently). But at least it doesn't point to a mutable object.

CHECK

EXPLAIN

Unhelpful 3

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.

READING EXERCISES

Implementing `stringValue`

Here is the recursive implementation of `stringValue()` with the recursive steps brought together but with the base case still missing:

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

```
/**  
 * @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. No unnecessary leading zeros are included.  
 */  
  
public static String stringValue(int n, int base) {  
    if (n < 0) {  
        return "-" + stringValue(-n, base);  
    } else if (BASE CONDITION) {  
        BASE CASE  
    } else {  
        return stringValue(n/base, base) + "0123456789".charAt(n%base);  
    }  
}
```

Which of the following can be substituted for the `BASE CONDITION` and `BASE CASE` to make the code correct?

- ✖ else if ($n == 0$) { return "0"; }
✖ else if ($n < base$) { return "" + n; } ↗
✓ else if ($n == 0$) { return ""; }
✖ else if ($n < base$) { return "0123456789".substring($n, n+1$); } ↗

➤ The first choice is wrong because it will add a leading 0 to single-digit numbers, i.e. making `stringValue(3, 10)` return "03" instead of just "3".

The second choice works. `return "" + n` is shorthand for converting the single-digit number n into a string.

The third choice is wrong because `stringValue(0, 10)` will return "" instead of "0".

The fourth choice works.

CHECK

EXPLAIN

Calling stringValue

Assuming the code is completed with one of the base cases identified in the previous problem, what does `stringValue(170, 16)` do?

- ✖ returns "AA"
✖ returns "170"
✖ returns "1010"
✓ throws `StringIndexOutOfBoundsException` ↗
✖ doesn't compile, static error
✓ StackOverflowError
✖ infinite loop

➤ Note first that using $base=16$ violates the precondition of this method, so it doesn't have to satisfy the postcondition. A valid implementation can do anything. The question is what this particular valid implementation will do.

The recursive step will be invoked, which will split the number 170 by computing $170/16=10$ and $170\%16=10$. The `charAt()` call will attempt to get the 11th character of "0123456789", which is past the end of the string. A `StringIndexOutOfBoundsException` will result.

CHECK

EXPLAIN

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.

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

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 can 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` (<http://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` (<http://docs.oracle.com/javase/8/docs/api/index.html?java/nio/file/Files.html>) and `java.nio.Path` (<http://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.

READING EXERCISES

```
subsequences("123456")
```

Recall the implementation of `subsequences()` from the start of this reading:

```
public static String subsequences(String word) {  
    if (word.isEmpty()) {  
        return ""; // base case  
    } else {  
        char firstLetter = word.charAt(0);  
        String restOfWord = word.substring(1);  
  
        String subsequencesOfRest = subsequences(restOfWord);  
  
        String result = "";  
        for (String subsequence : subsequencesOfRest.split(", ", -1)) {  
            result += "," + subsequence;  
            result += "," + firstLetter + subsequence;  
        }  
        if (result.startsWith(", ")) result = result.substring(1);  
        return result;  
    }  
}
```

For `subsequences("123456")`, how deep does its recursive call stack get? How many recursive calls to `subsequences()` can be active at the same time?



fd

7

➤ The calls are "123456", "23456", "3456", ..., "56", "6", "", for a total of 7.

CHECK

EXPLAIN

You are not logged in.

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.

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

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 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
Fix the Bug	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:

Reading 11: Debugging

Reproduce the Bug

Understand the Location and Cause of the Bug

Fix the Bug

```
/**  
 * 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.

READING EXERCISES

Reducing a bug to a test case

Regression testing

You are not logged in.

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.

Reading 11: Debugging

Reproduce the Bug

Understand the Location and Cause of the Bug

Fix the Bug

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

Reading 11: Debugging

Reproduce the Bug

Understand the Location and Cause of the Bug

Fix the Bug

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.

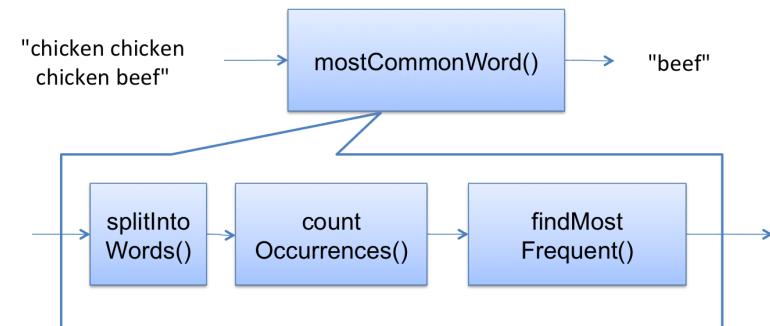
The process of isolating a small test case may also give you data that you didn't have before. You may even have two related test cases that *bracket* the bug in the sense that one succeeds and one fails. For example, maybe `mostCommonWords("c c, b")` is broken, but `mostCommonWords("c c b")` is fine.

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

Reading 11: Debugging

Reproduce the Bug

Understand the Location and Cause of the Bug

Fix the Bug

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.

Reading 11: Debugging

Reproduce the Bug

Understand the Location and Cause of the Bug

Fix the Bug

READING EXERCISES

Debugging strategy

Suppose you are debugging the `quadraticRoots` function, which appears to be producing wrong answers sometimes.

```
/**  
 * Solves quadratic equation ax^2 + bx + c = 0.  
 *  
 * @param a quadratic coefficient, requires a != 0  
 * @param b linear coefficient  
 * @param c constant term  
 * @return a list of the real roots of the equation  
 */  
public static List<Double> quadraticRoots(int a, int b, int c) { ... }
```

Put the following items in the order that you should try them: 1, 2, 3, ... Say "wat" for items that are nonsense statements.

Change your code from using `ArrayList` to using `LinkedList`.

✗ | 4

Put `println()` statements throughout your method to display the intermediate values of the calculation.

✗ | 3

Write a test case that causes the bug to happen.

✗ | 1

Run a code coverage tool to see if there are lines that your test suite wasn't reaching.

✗ | 2

Switch from Java 8 back to Java 7.



1



5

Reading 11: Debugging

Reproduce the Bug

Understand the Location and Cause of the Bug

Fix the Bug

Having a failing test case makes it much easier to debug, and is always the best thing to start with.

Running the code coverage tool is not a bad next step, because it's easy to run and may help localize the bug to untested code.

Inserting print statements to collect information takes more effort but the effort pays off.

Changing from `ArrayList` to `LinkedList`, or Java 8 to Java 7, are good examples of the “swap components” technique, but these are trusted components that are supposed to behave alike, so they would be the last things to try.

CHECK

EXPLAIN

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.

Collaboratively authored with contributions from:
Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama.
This work is licensed under CC BY-SA 4.0
(<http://creativecommons.org/licenses/by-sa/4.0/>).

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** (<http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>) in the Java Tutorials.

READING EXERCISES

The following questions use the code below. Study it first, then answer the questions.

You are not
logged in.

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

```
class Wallet {  
    private int amount;  
  
    public void loanTo(Wallet that) {  
        // put all of this wallet's money into that wallet  
        /*A*/ that.amount += this.amount;  
        /*B*/ amount = 0;  
    }  
  
    public static void main(String[] args) {  
        /*C*/ Wallet w = new Wallet();  
        /*D*/ w.amount = 100;  
        /*E*/ w.loanTo(w);  
    }  
}  
  
class Person {  
    private Wallet w;  
  
    public int getNetWorth() {  
        /*F*/ return w.amount;  
    }  
  
    public boolean isBroke() {  
        /*G*/ return Wallet.amount == 0;  
    }  
}
```

Access control A

Which of the following statements are true about the line marked `/*A*/` ?

`that.amount += this.amount;`

- The reference to `this.amount` is allowed by Java.
- The reference to `this.amount` is not allowed by Java because it uses `this` to access a private field.
- The reference to `that.amount` is allowed by Java.

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

The reference to `that.amount` is not allowed by Java because `that.amount` is a private field in a different object.

The reference to `that.amount` is not allowed by Java because it writes to a private field.

The illegal access(es) are caught statically.

The illegal access(es) are caught dynamically.

➤ Private fields and methods can be used by any code in the same class. `Wallet`'s private fields and methods can be used by any code in the `Wallet` class, even to access private fields in more than one `Wallet` object, not just `this`. Roughly speaking, any code within the curly braces of `Wallet`'s class body can touch `Wallet`'s private fields and methods. This isn't strictly true, because `Wallet` might contain nested class definitions that don't automatically get access to `Wallet`'s private fields and methods, but aside from that, it's a useful rule of thumb.

CHECK

EXPLAIN

Access control B

Which of the following statements are true about the line marked `/*B*/` ?

```
amount = 0;
```

✖ The reference to `amount` is allowed by Java.

The reference to `amount` is not allowed by Java because it doesn't use `this`.

The illegal access is caught statically.

The illegal access is caught dynamically.

➤ Private fields and methods can be used by any code in the same class. For fields, the `this` reference is implicit and can be omitted.

CHECK

EXPLAIN

Access control C

Which of the following statements are true about the line marked `/*C*/` ?

```
Wallet w = new Wallet();
```

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

- ✖ The call to the `Wallet()` constructor is allowed by Java.
- The call to the `Wallet()` constructor is not allowed by Java because there is no public `Wallet()` constructor declared.

- The illegal access is caught statically.
- The illegal access is caught dynamically.

- In Java, a class with no explicitly-declared constructors gets an automatic public constructor that takes no arguments.

CHECK

EXPLAIN

Access control D

Which of the following statements are true about the line marked `/*D*/` ?

```
w.amount = 100;
```

- ✖ The access to `w.amount` is allowed by Java.
- The access to `w.amount` is not allowed by Java because `amount` is private.
- The illegal access is caught statically.
- The illegal access is caught dynamically.

- `Wallet`'s private fields and methods can be used by any code in the `Wallet` class, even static methods. Roughly speaking, any code within the curly braces of `Wallet`'s class body can touch `Wallet`'s private fields and methods. This isn't strictly true because `Wallet` might contain nested class definitions that don't automatically get access to `Wallet`'s private fields and methods, but aside from that, it's a useful rule of thumb.

CHECK

EXPLAIN

Access control E

Which of the following statements are true about the line marked /*E*/

```
w.loanTo(w);
```

- ✓ The call to `loanTo()` is allowed by Java.
- The call to `loanTo()` is not allowed by Java because `this` and `that` will be aliases to the same object.
- The problem will be found by a static check.
- The problem will be found by a dynamic check.
- ✓ After this line, the `Wallet` object pointed to by `w` will have amount 0.
- After this line, the `Wallet` object pointed to by `w` will have amount 100.
- After this line, the `Wallet` object pointed to by `w` will have amount 200.
- In this call to `loanTo()`, `this` and `that` will indeed be aliases for the same object, but Java doesn't prevent it. It causes `loanTo()` to behave badly, emptying out the wallet.

CHECK

EXPLAIN

Access control F

Which of the following statements are true about the line marked /*F*/ ?

```
return w.amount;
```

- ✗ The reference to `w.amount` is allowed by Java because both `w` and `amount` are private variables.
- ✓ The reference to `w.amount` is allowed by Java because `amount` is a primitive type, even though it's private.
- The reference to `w.amount` is not allowed by Java because `amount` is a private field in a different class.
- ✗
- ✓ The illegal access is caught statically.
- The illegal access is caught dynamically.

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

- This code is not in the `Wallet` class, so it is not allowed to access `Wallet`'s private fields and methods. Java produces a static compiler error about it.

CHECK

EXPLAIN

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

Access control G

Which of the following statements are true about the line marked `/*G*/` ?

```
return Wallet.amount == 0;
```

- ✗ The reference to `Wallet.amount` is allowed by Java because `Wallet` has permission to access its own private field `amount`.
- The reference to `Wallet.amount` is allowed by Java because `amount` is a static variable.
- The reference to `Wallet.amount` is not allowed by Java because `amount` is a private field.
☒
- The reference to `Wallet.amount` is not allowed by Java because `amount` is an instance variable.
☒
- The illegal access is caught statically. ☒
- The illegal access is caught dynamically.

- `amount` is an instance variable, and requires a specific `Wallet` object instance to access. So you can't refer to it with the class name `Wallet` like you would a static variable, you have to use an instance object like `w` in the previous examples.
- `amount` is also private, so not accessible here.
- Both problems would have to be fixed – changing `amount` to `public static` – in order to make the code legal. But doing this would certainly be a bad idea, because it would make `amount` a global variable. Everybody would be able to look at and change your wallet. In fact there would only be a single `amount` variable, shared by everybody's wallet!

CHECK

EXPLAIN

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:

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

- **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.

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

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):

- creator: $t^* \rightarrow T$
- producer: $T+, t^* \rightarrow T$
- observer: $T+, t^* \rightarrow t$
- mutator: $T+, t^* \rightarrow void|t|T$

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. For example, a producer may take two values of the abstract type, like `String.concat()` does. The occurrences of `t` on the left may also be omitted, since some observers take no non-abstract arguments, and some take several.

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

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()` (<http://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` (<http://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()` (<http://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()` (<http://docs.oracle.com/javase/8/docs/api/java.awt/Container.html#add-java.awt.Component->) returns the object itself, so that multiple `add()` calls can be chained together (http://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` (<http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#singletonList-T->)
- producers: `Collections.unmodifiableList` (<http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#unmodifiableList-T->)
- observers: `size`, `get`
- mutators: `add`, `remove`, `addAll`, `Collections.sort` (<http://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.

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 EXERCISES

Operations

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.

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

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:

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

```
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?

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

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.

READING EXERCISES

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

Representation 1

Consider the following abstract data type.

```
/*
 * Represents a family that lives in a household together.
 * A family always has at least one person in it.
 * Families are mutable.
 */
class Family {
    // the people in the family, sorted from oldest to youngest, with no duplicates.
    public List<Person> people;

    /**
     * @return a list containing all the members of the family, with no duplicates.
     */
    public List<Person> getMembers() {
        return people;
    }
}
```

Here is a client of this abstract data type:

```
void client1(Family f) {
    // get youngest person in the family
    Person baby = f.people.get(f.people.size()-1);
    ...
}
```

Assume all this code works correctly (both `Family` and `client1`) and passes all its tests.

Now `Family`'s representation is changed from a `List` to `Set`, as shown:

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

```
/**  
 * Represents a family that lives in a household together.  
 * A family always has at least one person in it.  
 * Families are mutable.  
 */  
class Family {  
    // the people in the family  
    public Set<Person> people;  
  
    /**  
     * @return a list containing all the members of the family, with no duplicates.  
     */  
    public List<Person> getMembers() {  
        return new ArrayList<Person>(people);  
    }  
}
```

Assume that `Family` compiles correctly after the change.

Which of the following statements are true about `client1` after `Family` is changed?

- client1 is independent of `Family`'s representation, so it keeps working correctly.
 - client1 depends on `Family`'s representation, and the dependency would be caught as a static error.
 - client1 depends on `Family`'s representation, and the dependency would be caught as a dynamic error.
 - client1 depends on `Family`'s representation, and the dependency would not be caught but would produce a wrong answer at runtime.
 - client1 depends on `Family`'s representation, and the dependency would not be caught but would (luckily) still produce the same answer.
- client1 is directly accessing the `people` field in `Family`. When that field was a `List`, it could call `get()` on it with no trouble. Now that the field is a `Set`, the `get()` method doesn't exist, so there is a static error.

Because client1 no longer works when Family's representation changes, we say that client1 has a dependency on Family's representation.

CHECK

EXPLAIN

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

Representation 2

Now consider client2:

```
void client2(Family f) {  
    // get size of the family  
    int familySize = f.people.size();  
    ...  
}
```

Which of the following statements are true about client2 after Family is changed?

- client2 is independent of Family's representation, so it keeps working correctly.
 - client2 depends on Family's representation, and the dependency would be caught as a static error.
 - client2 depends on Family's representation, and the dependency would be caught as a dynamic error.
 - client2 depends on Family's representation, and the dependency would not be caught but would produce a wrong answer at runtime.
 - client2 depends on Family's representation, and the dependency would not be caught but would (luckily) still produce the same answer.
-
- client2 is also directly accessing the people field in Family. Both the List version and the Set version of that field have a `size()` method, and since the List had no duplicates, `size()` in both cases returns the correct size of the family.
- But we still say that client2 has a dependency on Family's representation; it just got lucky this time. If the type of people field had instead changed to `People[]`, then client2 would no longer work, because it needs to use `.length` instead of `size()`.

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

Representation 3

Now consider `client3`:

```
void client3(Family f) {  
    // get any person in the family  
    Person anybody = f.getMembers().get(0);  
    ...  
}
```

Which of the following statements are true about `client3` after `Family` is changed?



- `client3` is independent of `Family`'s representation, so it keeps working correctly.
- `client3` depends on `Family`'s representation, and the dependency would be caught as a static error.
- `client3` depends on `Family`'s representation, and the dependency would be caught as a dynamic error.
- `client3` depends on `Family`'s representation, and the dependency would not be caught but would produce a wrong answer at runtime.
- `client3` depends on `Family`'s representation, and the dependency would not be caught but would (luckily) still produce the same answer.



`client3` is calling a public method to get the members of the family, and it depends only on the contract of that public method. Note that the contract of `getMembers()` doesn't say anything about the order of the people in the list it returns, but `client3` doesn't care about that ordering anyway. `client3` only cares that the list has at least one person in it, and `Family`'s contract as a whole promises that.

Since `getMembers()` still satisfies its contract, even with the new `Set` representation, `client3` will keep working after the change. It is independent of the representation.

Representation 4

Reading 12: Abstract Data Types

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

For each section of the Family data type's code shown below, is it part of the ADT's specification, its representation, or its implementation?



specification

```
/**  
 * Represents a family that lives in a household together.  
 * A family always has at least one person in it.  
 * Families are mutable.  
 */
```



specification

```
public class Family {
```



representation

```
// the people in the family, sorted from oldest to  
youngest, with no duplicates.
```



implementation

```
private List<Person> people;
```



implementation

```
/**  
 * @return a list containing all the members of the family, with no duplicates.  
 */
```



representation

```
public List<Person> getMembers() {
```



specification

```
return people;
```

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

```
}
```

➤ The specification of an ADT includes the name of the class (on line 6), the Javadoc comment just before the class (lines 1-5), and the specifications of its public methods and fields (Javadoc lines 10-12 and method signature line 13). These parts are the contract that is visible to a client of the class.

The representation of an ADT consists of its fields (line 8) and any assumptions or requirements about those fields (line 7).

The implementation of an ADT consists of the method implementations that manipulate its rep (line 14).

CHECK

EXPLAIN

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	ArrayList() (http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#ArrayList--)
	Static (factory) method	Collections.singletonList() (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#singletonList-T-), Arrays.asList() (http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList-T...-)
	Constant	BigInteger.ZERO (http://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html#ZERO)
	Observer operation	List.get() (http://docs.oracle.com/javase/8/docs/api/java/util/List.html#get-int-)

Reading 12: Abstract Data Types	Static method	Collections.max() (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#max--java.util.Collection-)	
What Abstraction Means	Producer operation	String.trim() (http://docs.oracle.com/javase/8/docs/api/java/lang/String.html#trim--)	
Classifying Types and Operations	Static method	Collections.unmodifiableList() (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#unmodifiableList--java.util.List-)	
Designing an Abstract Type	Mutator operation	Instance method	List.add() (http://docs.oracle.com/javase/8/docs/api/java/util/List.html#add-E-)
Representation Independence	Static method	Collections.copy() (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#copy--java.util.List-javax.util.List-)	
Realizing ADT Concepts in Java	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:

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

```
// 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:

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

```
@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());
    }
```

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

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.

READING EXERCISES

These questions use the following datatype:

```
/** Immutable datatype representing a student's progress through school. */
class Student {

    /** make a freshman */
    public Student() { ... }

    /** @return a student promoted to the next year, i.e.
        freshman returns a sophomore,
        sophomore returns a junior,
        junior returns a senior,
        senior returns an alum,
        alum stays an alum and can't be promoted further. */
    public Student promote() { ... }

    /** @return number of years of school completed, i.e.
        0 for a freshman, 4 for an alum */
    public int getYears() { ... }

}
```

Partitioning ADT operations

How many parts are there in a reasonable input-space partition of the `Student()` constructor?



0



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

Collaboratively authored with contributions from:
Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama.

This work is licensed under CC BY-SA 4.0
(<http://creativecommons.org/licenses/by-sa/4.0/>).

How many parts are there in a reasonable, but not exhaustive, input-space partition of `promote()`?

How many parts are there in a reasonable, but not exhaustive, input-space partition of `getYears()`?

CHECK

EXPLAIN

Choosing ADT test cases

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.

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

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

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;  
    }  
}
```

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

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:

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

```
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`:

Reading 13: Abstraction Functions & Rep Invariants

Invariants

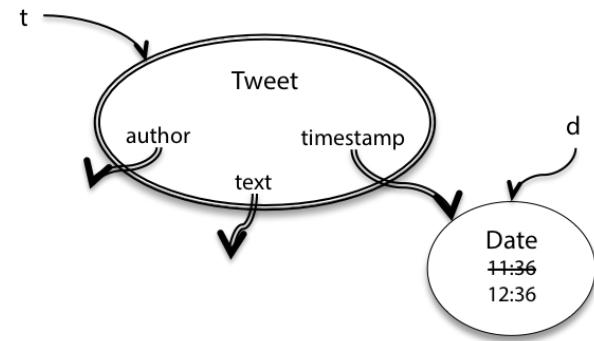
Rep Invariant and Abstraction Function

Documenting the AF, RI, and Safety from Rep Exposure

ADT invariants replace preconditions

Summary

```
/** @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(Date.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* (<http://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:

Reading 13: Abstraction Functions & Rep Invariants

Invariants

Rep Invariant and Abstraction Function

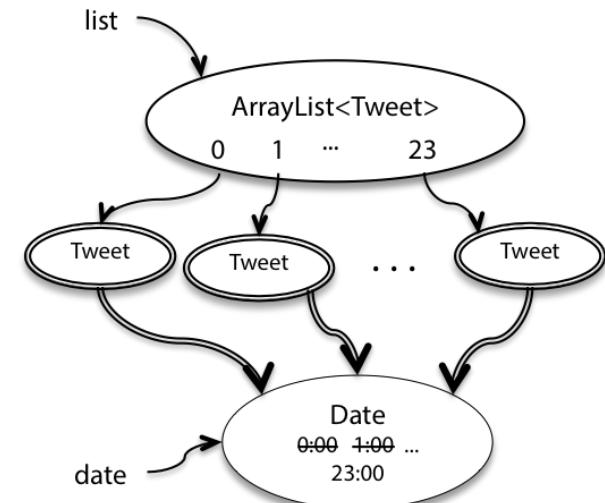
Documenting the AF, RI, and Safety from Rep Exposure

ADT invariants replace preconditions

Summary

```
/** @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?

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

```
/**  
 * 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 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()` throw exceptions. So you can construct a list using mutators, then seal it up in an unmodifiable wrapper (and throw away your reference to the original mutable list), and get an immutable list.

The downside here is that you get immutability at runtime, but not at compile time. Java won't warn you at compile time if you try to `sort()` this unmodifiable list. You'll just get an exception at runtime. 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.

READING EXERCISES

Rep exposure

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

Consider the following problematic datatype:

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

```
/** Represents an immutable right triangle. */
class RightTriangle {
    /*A*/    private double[] sides;

        // sides[0] and sides[1] are the two legs,
        // and sides[2] is the hypotenuse, so declare it to avoid ha
ving a
        // magic number in the code:
    /*B*/    public static final int HYPOTENUSE = 2;

        /** Make a right triangle.
         * @param legA, legB  the two legs of the triangle
         * @param hypotenuse  the hypotenuse of the triangle.
         *          Requires hypotenuse^2 = legA^2 + legB^2
         *          (within the error tolerance of double arithmetic)
    /*C*/
        /*
         * @param legA, legB, hypotenuse
         * @return RightTriangle with sides legA, legB, hypotenuse
    /*D*/
        public RightTriangle(double legA, double legB, double hypote
nuse) {
            this.sides = new double[] { legA, legB, hypotenuse };
        }

        /** Get all the sides of the triangle.
         * @return three-element array with the triangle's side lengths
        /*
         * @return sides
    /*E*/
        public double[] getAllSides() {
            return sides;
        }

        /** @return length of the triangle's hypotenuse */
        public double getHypotenuse() {
            return sides[HYPOTENUSE];
        }

        /** @param factor to multiply the sides by
         * @return a triangle made from this triangle by
         * multiplies all side lengths by factor.
        /*

```

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

```
public RightTriangle scale(double factor) {
    return new RightTriangle (sides[0]*factor, sides[1]*factor,
                           sides[2]*factor);
}

/** @return a regular triangle made from this triangle.
 * A regular right triangle is one in which
 * both legs have the same length.
 */
public RightTriangle regularize() {
    double bigLeg = Math.max(side[0], side[1]);
    return new RightTriangle (bigLeg, bigLeg, side[2]);
}
```

Which of the following statements are true?

- The line marked /*A*/ is a problem for rep exposure because arrays are mutable.
- The line marked /*B*/ is a problem for representation independence because it reveals how the sides array is organized.
- The line marked /*C*/ is a problem because creator operations should not have preconditions.
- The line marked /*D*/ is a problem because it puts legA, legB, and hypotenuse into the rep without doing a defensive copy first.
- The line marked /*E*/ is a problem because it threatens the class's immutability.

CHECK

EXPLAIN

Rep Invariant and Abstraction Function

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

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.

Reading 13: Abstraction Functions & Rep Invariants

Invariants

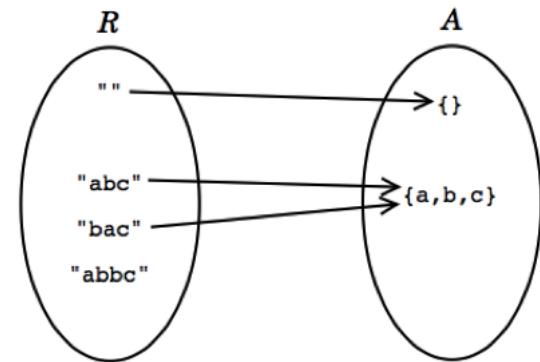
Rep Invariant and Abstraction Function

Documenting the AF, RI, and Safety from Rep Exposure

ADT invariants replace preconditions

Summary

- **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 bijective (also called *one-to-one*), 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.

Both the rep invariant and the abstraction function should be documented in the code, right next to the declaration of the rep itself:

Reading 13: Abstraction Functions & Rep Invariants

Invariants

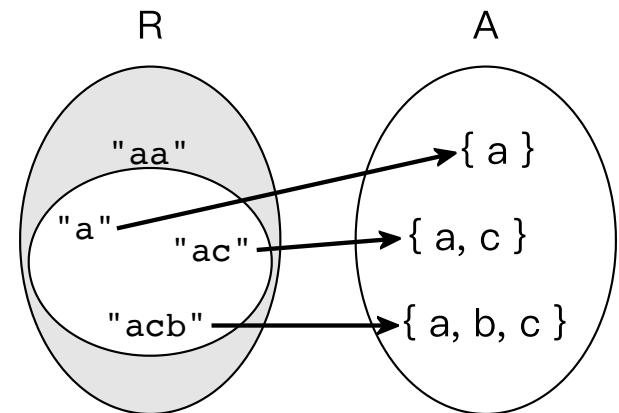
Rep Invariant and Abstraction Function

Documenting the AF, RI, and Safety from Rep Exposure

ADT invariants replace preconditions

Summary

```
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  
    ...  
}
```

Reading 13: Abstraction Functions & Rep Invariants

Invariants

Rep Invariant and Abstraction Function

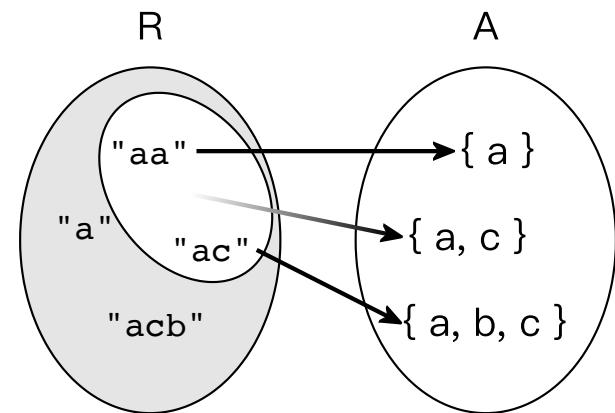
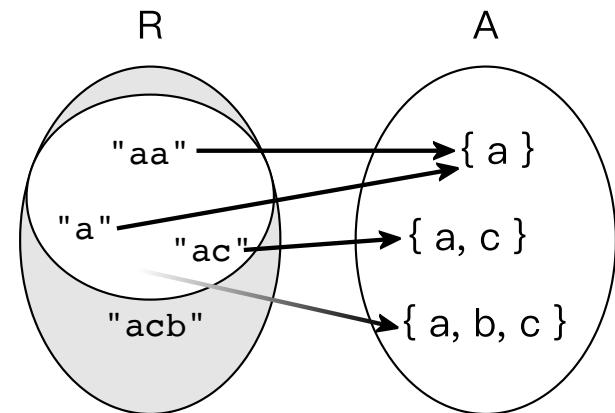
Documenting the AF, RI, and Safety from Rep Exposure

ADT invariants replace preconditions

Summary

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:

```
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?

READING EXERCISES

Who knows what?

Which of the following should be known (visible and documented) to the client of an abstract data type?

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

- ✗ abstract value space
- abstraction function
- creators
- observers
- rep
- rep invariant

➤ In order to preserve representation independence, the client should know about things related to the abstraction, but not about the rep. From this list, the abstraction includes the abstract value space (e.g., CharSet's abstract value space is "sets of characters"), creators (e.g., CharSet's constructor), observers (e.g., CharSet.size()). But the rep, rep invariant, and abstraction function involve knowledge of the rep, so they should not generally be known to the client.

Which of the following should be known to the maintainer of an abstract data type?

- ✗ abstract value space
- abstraction function
- creators
- observers
- rep
- rep invariant

➤ The maintainer of an abstract data type has to know about both the abstraction and the rep, so all elements should be known.

CHECK

EXPLAIN

Rep invariant pieces

Suppose C is an abstract data type whose representation has two String fields:

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

```
class C {  
    private String s;  
    private String t;  
    ...  
}
```

Assuming you don't know anything about C's abstraction, which of the following might be statements in a rep invariant for C?

- ✖ s contains only letters
- s.length() == t.length()
- s represents a set of characters
- C's observers
- s is the reverse of t
- s+t

➤ Recall that the rep invariant is a function from rep values (pairs of `String` objects `s, t`) to boolean, so the only good answers to this question are boolean predicates (true or false statements) that constrain legal values of `s` and `t`.

“s represents a set of characters” might belong in an abstraction function, but not in a rep invariant.

C's observer operations are part of the abstraction, not the rep.

`s+t` is not a boolean predicate.

CHECK

EXPLAIN

Trying to implement without an AF/RI

Suppose Louis Reasoner has created `CharSet` with the following rep:

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

```
public class CharSet {  
    private String s;  
    ...  
}
```

But Louis unfortunately neglects to write down the abstraction function (AF) and rep invariant (RI). Here are four possible AF/RI pairs that *might* have been what Louis had in mind. All of them were also mentioned in the reading above.

SortedRep:

```
// AF: represents the set of characters found in s  
// RI: s[0] < s[1] < ... < s[s.length()-1]
```

SortedRangeRep:

```
// AF: represents the union of the ranges {s[i]...s[i+1]} for each adjacent pair of characters in s  
// RI: s.length is even, and s[0] < s[1] < ... < s[s.length()-1]
```

NoRepeatsRep:

```
// AF: represents the set of characters found in s  
// RI: s contains no character more than once
```

AnyRep:

```
// AF: represents the set of characters found in s  
// RI: true
```

Louis has three teammates helping him implement CharSet, each working on a different operation: add(), remove(), and contains(). Their implementations are below. Which of the possible AF/RI pairs are consistent with each programmer's implementation?

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

```
public void add(char c) {  
    s = s + c;  
}
```

- ✗ SortedRep
- SortedRangeRep
- NoRepeatsRep
- AnyRep ✕

➤ The programmer who wrote `add()` did it the easiest way possible.

It isn't consistent with `SortedRep` or `SortedRangeRep`, because those reps require the character to be put in a particular place, depending on the value of `c`, not just at the end of the string.

It isn't consistent with `NoRepeatsRep` because `c` may already occur in the string, and `add()` isn't checking with that.

But it is consistent with `AnyRep`, whose RI allows any string of characters and whose AF interprets the string in such a way that `c` is considered part of the resulting set.

```
public void remove(char c) {  
    int position = s.indexOf(c);  
    if (position >= 0) {  
        s = s.substring(0, position) + s.substring(position+1, s.length  
());  
    }  
}
```

- ✗ SortedRep ✕
- SortedRangeRep
- NoRepeatsRep ✕
- AnyRep

➤ This implementation of `remove()` finds the first occurrence of `c` in the string and removes it.

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

This is consistent with SortedRep, because it still keeps the string ordered. Note also that if you read SortedRep's RI carefully, you'll see that it also forbids duplicates, so we're guaranteed that the string won't have any other occurrences of `c` that we need to remove.

It's not consistent with SortedRangeRep, because it will make an even-length string into an odd-length string, by throwing away one end of a range pair.

It is consistent with NoRepeatsRep, because that rep invariant guarantees the string will have at most one occurrence of `c` to remove.

It is not consistent with AnyRep, because of the possibility that `c` is duplicated in the string. If the string is "xxyx" and we remove just the first 'x', the string will become "yyx". We'll have failed to remove 'x' from the set.

```
public boolean contains(char c) {  
    for (int i = 0; i < s.length(); i += 2) {  
        char low = s.charAt(i);  
        char high = s.charAt(i+1);  
        if (low <= c && c <= high) {  
            return true;  
        }  
    }  
    return false;  
}
```

- ✖ SortedRep
- SortedRangeRep ✅
- NoRepeatsRep
- AnyRep

➤ This version of `contains()` strongly assumes the SortedRange rep. It may throw an exception sometimes with the other reps, when it reaches the end of an odd-length string and `s.charAt(i+1)` tries to access beyond the end of the string.

CHECK

EXPLAIN

Example: Rational Numbers

Here's an example of an abstract data type for rational numbers. Look closely at its rep invariant and abstraction function.

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

Invariants

Rep Invariant and Abstraction Function

Documenting the AF, RI, and Safety from Rep Exposure

ADT invariants replace preconditions

Summary

```
public class RatNum {  
    private final int numer;  
    private final int denom;  
  
    // Rep invariant:  
    //   denom > 0  
    //   numer/denom is in reduced form  
  
    // Abstraction Function:  
    //   represents the rational number numer / denom  
  
    /** Make a new Ratnum == n. */  
    public RatNum(int n) {  
        numer = n;  
        denom = 1;  
        checkRep();  
    }  
  
    /**  
     * Make a new RatNum == (n / d).  
     * @param n numerator  
     * @param d denominator  
     * @throws ArithmeticException if d == 0  
     */  
    public RatNum(int n, int d) throws ArithmeticException {  
        // reduce ratio to lowest terms  
        int g = gcd(n, d);  
        n = n / g;  
        d = d / g;  
  
        // make denominator positive  
        if (d < 0) {  
            numer = -n;  
            denom = -d;  
        } else {  
            numer = n;  
            denom = d;  
        }  
        checkRep();  
    }  
}
```

```
}
```

Reading 13: Abstraction Functions & Rep Invariants

Invariants

Rep Invariant and Abstraction Function

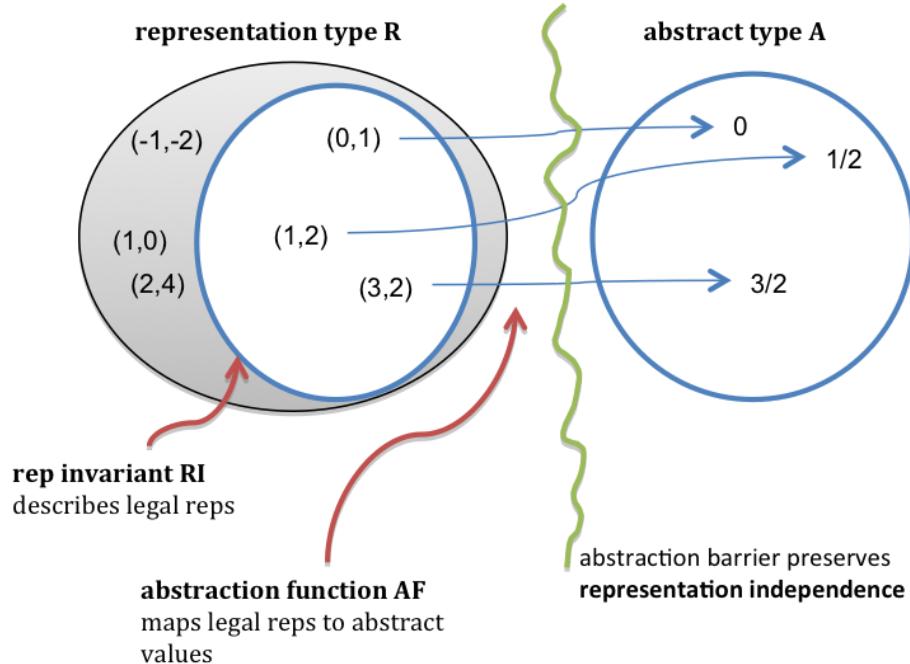
Documenting the AF, RI, and Safety from Rep Exposure

ADT invariants replace preconditions

Summary

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;
}
```

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

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 specs 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;  
    ...  
}
```

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 EXERCISES

Checking the rep invariant

Which of the following are true?

- checkRep() is the abstraction function
 - checkRep() asserts the rep invariant
 - it's good for an implementer to call checkRep() just before returning from a public method of an ADT class
 -
 - it's good for a client to call checkRep() just after calling a public method of an ADT class
- checkRep() asserts the rep invariant, and it should be private and called within the class (i.e. just before returning), not by clients (just after calling).

CHECK

EXPLAIN

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:

**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

```
// 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.

Here are the arguments for `RatNum`.

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

```
// 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.

READING EXERCISES

Arguing against rep exposure

Consider the following ADT:

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

```
// Mutable type representing Twitter users' followers.  
public class FollowGraph {  
    private final Map<String,Set<String>> followersOf;  
  
    // Rep invariant:  
    //   all Strings in followersOf are Twitter usernames  
    //       (i.e., nonempty strings of letters, digits, underscore  
    s)  
    //   no user follows themselves, i.e. x is not in followersOf.get  
    (x)  
    // Abstraction function:  
    //   represents the follower graph where Twitter user x is followed  
    by user y  
    //       if and only if followersOf.get(x).contains(y)  
    // Safety from rep exposure:  
    //   All fields are private, and ???  
  
    // Operations (specs and method bodies omitted to save space)  
    public FollowGraph() { ... }  
    public void addFollower(String user, String follower) { ... }  
    public void removeFollower(String user, String follower) { ... }  
    public Set<String> getFollowers(String user) { ... }  
}
```

Assuming the omitted method bodies are consistent with the statement, which statement could replace ??? to make a persuasive safety-from-rep-exposure comment?

- ✓ "Strings are immutable."
- This is true, but insufficient, because the rep also contains mutable Map and Set objects.
- ✗ "followersOf is a mutable Map containing mutable Set objects, but getFollowers() makes a defensive copy of the Set it returns, and all other parameters and return values are immutable String or void."
- ☒

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

- This is a good argument. It considers each element of the rep (each private field, including all objects in the data structure that the field points to), and how each operation might affect it.
- ✓ "This class is mutable, so rep exposure isn't an issue."
- Immutability is not the only invariant that can be threatened by rep exposure. `FollowGraph` has a rep invariant that can be threatened if a `Set` of followers is inadvertently shared with a client.
- ✗ " `followersOf` is a mutable Map, but it is never passed or returned from an operation."
- The `Map` is not the only mutable type in the rep.
- ✗ " `FollowGraph()` does not expose the rep; `addFollower()` does not expose the rep; `removeFollower()` does not expose the rep; `getFollowers()` does not expose the rep."
- Proof by repeated assertion (https://en.wikipedia.org/wiki/Proof_by_assertion) is not an argument.
- ✗ " `String` is immutable, and the `Set` objects in the rep are made immutable by unmodifiable wrappers. The `Map` type is mutable, but that type is never passed or returned from an operation."
- This is a good argument. It considers all the types in the rep, asks whether they are immutable or not, and whether there is a static guarantee (for `Map` and `String`) or a dynamic check (for `Set`) that protects them from rep exposure. One could ask which approach is more error-prone – defensive copying, or making sure the sets stay unmodifiable internally – but the safety argument is nevertheless sound.

CHECK

EXPLAIN

How to Establish Invariants

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

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.

READING EXERCISES

Structural induction

Recall this data type from the first exercise in this reading:

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

```
/** Represents an immutable right triangle. */
class RightTriangle {
    private double[] sides;

    // sides[0] and sides[1] are the two legs,
    // and sides[2] is the hypotenuse, so declare it to avoid having a
    // magic number in the code:
    public static final int HYPOTENUSE = 2;

    /** Make a right triangle.
     * @param legA, legB  the two legs of the triangle
     * @param hypotenuse  the hypotenuse of the triangle.
     *         Requires hypotenuse^2 = legA^2 + legB^2
     *         (within the error tolerance of double arithmetic)
     */
    public RightTriangle(double legA, double legB, double hypotenuse) {
        this.sides = new double[] { legA, legB, hypotenuse };
    }

    /** Get all the sides of the triangle.
     * @return three-element array with the triangle's side lengths
     */
    public double[] getAllSides() {
        return sides;
    }

    /** @return length of the triangle's hypotenuse */
    public double getHypotenuse() {
        return sides[HYPOTENUSE];
    }

    /** @param factor to multiply the sides by
     * @return a triangle made from this triangle by
     * multiplies all side lengths by factor.
     */
    public RightTriangle scale(double factor) {
        return new RightTriangle (sides[0]*factor, sides[1]*factor, sides[2]*factor);
    }
}
```

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

```
/** @return a regular triangle made from this triangle.  
 * A regular right triangle is one in which  
 * both legs have the same length.  
 */  
public RightTriangle regularize() {  
    double bigLeg = Math.max(side[0], side[1]);  
    return new RightTriangle (bigLeg, bigLeg, side[2]);  
}  
}
```

This datatype has an important invariant: the relationship between the legs and hypotenuse, as stated in the Pythagorean theorem.

Assuming the client obeys the contracts when using RightTriangle, which of the following statements are true about this invariant?

- ✖ The creator RightTriangle() establishes the invariant if the client obeys all contracts.
☑
- If the client obeys the contract of the constructor, particularly its precondition, then the three sides of the triangle stored in the sides array satisfy the Pythagorean theorem invariant, as desired.
- ✖ The observer getAllSides() preserves the invariant if the client obeys all contracts.
- This method causes a rep exposure, so the client may inadvertently change values in the returned array and destroy the invariant as a result, even while obeying all the specs as written.
- ✖ The observer getHypotenuse() preserves the invariant if the client obeys all contracts.
☑
- This method doesn't mutate anything, and doesn't expose the rep, so if the invariant was true before getHypotenuse() is called, then it continues to be true afterward.

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

- ✖ The producer `scale()` preserves the invariant if the client obeys all contracts.
- This method creates a new triangle, multiplying each side by the same amount, so the new triangle is still a right triangle satisfying the Pythagorean theorem.
- ✓ The producer `regularize()` preserves the invariant if the client obeys all contracts.
- This method creates a new triangle, but changes only the legs and doesn't recalculate a new hypotenuse for it. This doesn't preserve the Pythagorean invariant for the new triangle.

CHECK

EXPLAIN

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);
```

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

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` type.

Many of the places where we used preconditions on the problem sets would have benefited from a custom ADT instead.

READING EXERCISES

Encapsulating preconditions in ADTs

Consider this method:

```
/**  
 * Find tweets written by a particular user.  
 *  
 * @param tweets a list of tweets with distinct timestamps, not modified  
 * by this method.  
 * @param username Twitter username (a nonempty sequence of letters, dig  
 * its, and underscore)  
 * @return all and only the tweets in the list whose author is username,  
 *         in the same order as in the input list.  
 */  
public static List<Tweet> writtenBy(List<Tweet> tweets, String username)  
{ ... }
```

You are not logged in.

Which ADTs would you create to eliminate the preconditions of this method?

- ✖ TweetsAndUsername
 - TweetList
 - Username
 - UsernameCharacter
- TweetList would be able to represent the requirement that the tweets have distinct timestamps, and Username would be able to represent the constraint on valid usernames.

CHECK

EXPLAIN

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

Collaboratively authored
Summary
with contributions from:
Saman Amarasinghe,
Adam Chlipala, Srinivasa
Devadas, Michael Ernst,
Max Goldman, John
Guttag, Daniel Jackson,
Rob Miller, Martin
Rinard, and Armando
Solar-Lezama. This work
is licensed under CC BY-
SA 4.0
(<http://creativecommons.org/licenses/by-sa/4.0/>).
Client code.

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.

Reading 14: Interfaces**Interfaces****Example: MyString****Example: Set****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* (http://web.mit.edu/6.005/www/sp15/classes/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.

Reading 14: Interfaces

Interfaces

Example: MyString

Example: Set

Why Interfaces?

Realizing ADT Concepts in Java

Summary

In the Java Tutorials, read these pages:

- **Defining an Interface** (<http://docs.oracle.com/javase/tutorial/java/landl/interfaceDef.html>)
- **Implementing an Interface**
(<http://docs.oracle.com/javase/tutorial/java/landl/usinginterface.html>)
- **Using an Interface as a Type**
(<http://docs.oracle.com/javase/tutorial/java/landl/interfaceAsType.html>)

READING EXERCISES

Java interfaces

Consider this Java interface and Java class, which are intended to implement an immutable set data type:

```
/** Represents an immutable set of elements of type E. */
interface Set<E> {
    /** make an empty set */
    A    public Set();
    /** @return true if this set contains e as a member */
    public boolean contains(E e);
    /** @return a set which is the union of this and that */
    B    public ArraySet union(Set<E> that);
}

/** Implementation of Set<E>. */
class ArraySet<E> implements Set<E> {
    /** make an empty set */
    public ArraySet() { ... }
    /** @return a set which is the union of this and that */
    public ArraySet union(Set<E> that) { ... }
    /** add e to this set */
    public void add(E e) { ... }
}
```

Which of the following statements are true about Set<E> and ArraySet<E> ?

The line labeled A is a problem because Java interfaces can't have constructors.

- True False

➤ Java interfaces can't have constructors, so there would be a static error on this line.

The line labeled B is a problem because Set mentions ArraySet , but ArraySet also mentions Set , which is circular.

- True False

➤ Circular references between classes and interfaces are legal in Java, and sometimes necessary.

The line labeled B is a problem because it isn't representation-independent.

- True False

➤ Returning ArraySet makes all clients of Set aware of, and possibly dependent on, the ArraySet implementation. The union operation should return a Set , not an ArraySet .

ArrayList doesn't correctly implement Set because it's missing the contains() method.

- True False

➤ ArrayList needs to implement all the methods of Set in order to satisfy Java's requirement for implements Set . The fact that it doesn't have contains() will cause a static error.

ArrayList doesn't correctly implement Set because it includes a method that Set doesn't have.

- True False

➤ Java does allow classes to have more methods than the interface they're implementing. Additional methods mean the class has a stronger spec than the interface, which is okay from the point of view of a client of the interface.

ArrayList doesn't correctly implement Set because ArrayList is mutable while Set is immutable.

- True False

➤ ArrayList has a mutator (add), which makes it a mutable class. This violates an important part of the spec for Set , its immutability, so ArrayList is not a legal implementation of Set . Java won't catch this problem, however. Neither a static error nor a dynamic error will be raised automatically.

Reading 14: Interfaces

Interfaces

Example: MyString

Example: Set

Why Interfaces?

Realizing ADT Concepts
in Java

Summary

Reading 14: Interfaces

Interfaces

Example: MyString

Example: Set

Why Interfaces?

Realizing ADT Concepts
in Java

Summary

Example: MyString

Let's revisit MyString (http://web.mit.edu/6.005/www/sp15/classes/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 (http://web.mit.edu/6.005/www/fa15/classes/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:

Reading 14: Interfaces

Interfaces

Example: MyString

Example: Set

Why Interfaces?

**Realizing ADT Concepts
in Java**

Summary

```

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;
    }
}

```

Reading 14: Interfaces
Interfaces
Example: MyString
Example: Set
Why Interfaces?
**Realizing ADT Concepts
in Java**
Summary

- Compare these classes to the implementations of `MyString` in *Abstract Data Types* (http://web.mit.edu/6.005/www/sp15/classes/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.

Reading 14: Interfaces

Interfaces

Example: MyString

Example: Set

Why Interfaces?

Realizing ADT Concepts in Java

Summary

- 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 provided them by default; adding the constructors that take `boolean b` means we have to declare the other constructors explicitly.

Now that we know good ADTs scrupulously preserve their own invariants (<http://web.mit.edu/6.005/www/fa15/classes/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);
    }

    // ...
}

```

Reading 14: Interfaces

Interfaces

Example: MyString

Example: Set

Why Interfaces?

Realizing ADT Concepts
in Java

Summary

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));

```

READING EXERCISES

Code review

Let's review the code for `FastMyString`. Which of these are useful criticisms:

I wish the abstraction function was documented

True False

> Every ADT must document the AF.

I wish the representation invariant was documented

True False

> Every ADT must document the RI and how it is preserved.

I wish the rep fields were `final` so they could not be reassigned

True False

> This ADT is immutable, so `final` fields would help preserve its (currently undocumented!) invariants

I wish the private constructor was public so clients could use it to construct empty strings

True False

Reading 14: Interfaces

Interfaces

Example: MyString

Example: Set

Why Interfaces?

Realizing ADT Concepts in Java

Summary

➤ Pointing out that clients can't currently make empty strings is quite valid. But the private constructor doesn't make empty strings — it makes instances that don't represent any value at all. We can't allow clients to see such `FastMyString` instances, so this constructor needs to stay private.

If we make the `rep` fields final, this constructor won't compile any more, and that's a sign that we need to improve the code to do a better job of preserving the invariants.

I wish the `charAt` specification did not expose that the `rep` contains individual characters

- True False

➤ The `rep` is not exposed; the abstract notion of string is a sequence of individual characters.

I wish the `charAt` implementation behaved more helpfully when `i` is greater than the length of the string

- True False

➤ If `a.length <= start + i`, the method will throw an exception. But for `end <= start + i < a.length`, we return the wrong answer! The implementation should fail fast.

CHECK

EXPLAIN

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:

Reading 14: Interfaces

Interfaces

Example: MyString

```
// 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.

Why Interfaces?

Realizing ADT Concepts in Java

Summary

```
// 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:

Reading 14: Interfaces

Interfaces

Example: MyString

Example: Set

Why Interfaces?

Realizing ADT Concepts
in Java

Summary

- Lesson: Interfaces (<http://docs.oracle.com/javase/tutorial/collections/interfaces/>)
- The Set Interface (<http://docs.oracle.com/javase/tutorial/collections/interfaces/set.html>)
- Set Implementations
(<http://docs.oracle.com/javase/tutorial/collections/implementations/set.html>)
- The List Interface (<http://docs.oracle.com/javase/tutorial/collections/interfaces/list.html>)
- List Implementations
(<http://docs.oracle.com/javase/tutorial/collections/implementations/list.html>)

READING EXERCISES

Collection interfaces & implementations

Assume the following lines of code are run in sequence, and that any lines of code that don't compile are simply commented out so that the rest of the code can compile.

The code uses two methods from Collections (<http://docs.oracle.com/javase/8/docs/api/?java/util/Collections.html>), so you might need to consult their documentation.

Choose the most specific answer to each question.

```
Set<String> set = new HashSet<String>();
```

set now points to:



an object that implements the Set interface

```
set = Collections.unmodifiableSet(set);
```

set now points to:



a Collections object

```
set = Collections.singleton("glorp");
```

set now points to:



null

You are not logged in.

```
set = new Set<String>();
```

Reading 14: Interfaces

Interfaces

Example: MyString

Example: Set

Why Interfaces?

Realizing ADT Concepts in Java

Summary

set now points to:

✗ | an object that implements the Set interface ▾

```
List<String> list = set;
```

set now points to:

✗ | an object that implements the Set interface ▾

CHECK

EXPLAIN

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,

Reading 14: Interfaces

Interfaces

Example: MyString

Example: Set

Why Interfaces?

Realizing ADT Concepts
in Java

Summary

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 (http://web.mit.edu/6.005/www/fa15/classes/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> (http://docs.oracle.com/javase/8/docs/api/java/lang/String.html)
	Interface + class(es)	<code>List</code> (http://docs.oracle.com/javase/8/docs/api/java/util/List.html) and <code>ArrayList</code> (http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html)
Creator operation	Constructor	<code>ArrayList()</code> (http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#ArrayList--)
	Static (factory) method	<code>Collections.singletonList()</code> (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#singletonList-T-), <code>Arrays.toList()</code> (http://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#asList-T...-)
	Constant	<code>BigInteger.ZERO</code> (http://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html#ZERO)
	Instance method	<code>List.get()</code> (http://docs.oracle.com/javase/8/docs/api/java/util/List.html#get-int-)
Observer operation	Static method	<code>Collections.max()</code> (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#max-java.util.Collection-)

Reading 14: Interfaces	Producer operation	Instance method	<code>String.trim()</code> (http://docs.oracle.com/javase/8/docs/api/java/lang/String.html#trim--)
Interfaces		Static method	<code>Collections.unmodifiableList()</code> (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#unmodifiableList--java.util.List-)
Example: MyString	Mutator operation	Instance method	<code>List.add()</code> (http://docs.oracle.com/javase/8/docs/api/java/util/List.html#add-E-)
Example: Set		Static method	<code>Collections.copy()</code> (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#copy--java.util.List-javax.util.List-)
Why Interfaces?			
Realizing ADT Concepts in Java	Representation	private fields	
Summary			

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.

Collaboratively authored with contributions from:
 Saman Amarasinghe, Adam Chlipala, Srini Devadas,
 Michael Ernst, Max Goldman, John Gutttag,
 Daniel Jackson, Rob Miller,
 Martin Rinard, and Armando Solar-Lezama.
 This work is licensed under CC BY-SA 4.0
 (<http://creativecommons.org/licenses/by-sa/4.0/>).

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:

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

- $|\{1,2\}| = 2$ 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.

READING EXERCISES

Any second now

Consider the code for `Duration` and the objects `d1`, `d2`, `d3`, `d4` just created above.

Using the abstraction-function notion of equality, which of the following would be considered equal to `d1`?

- d1
- d2
- d3
- d4

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

► The abstraction function maps `d1`, `d3`, and `d4` to the same span of time, 62 seconds. But it maps `d2` to a different span of time, 63 seconds.

CHECK

EXPLAIN

Eye on the clock

Using the observational notion of equality, which of the following would be considered equal to `d1`?

- ✗ d1 ✗
 d2
✓ d3 ✗
 d4 ✗

► The only operation that the Duration ADT offers is `length()`, and `d1`, `d3`, and `d4` all return the same result, 62 seconds. But `d2` is distinguishable from `d1` using the `length()` operation, so it would not be equal to the others under an observational interpretation of equality.

CHECK

EXPLAIN

== 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:

	<i>referential equality</i>	<i>object equality</i>
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:

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

```
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

([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\(Duration\)`.](http://www.pythontutor.com/java.html#code=public+class+Duration+%7B%0A++++private+final+int+mins%3B%0A++++private+final+int+secs%3B%0A++++//rep+invariant%3A%0A++++/++++mins+%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++++System.out.println(%22d1.equals(o2)%22frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputJSON=%5B%5D&curlInstr=33). You'll see that even though <code>d2</code> and <code>o2</code> end up referring to the very same object in memory, you still get different results for them from <code>equals()</code>.</p></div><div data-bbox=)

```
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 (<http://web.mit.edu/6.005/www/fa15/classes/01-static-checking/#types>).

Recall from the Java Tutorials (<http://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.

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

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

(<a href="http://www.pythontutor.com/java.html#code=public+class+Duration+%7B%0A++++private+final+int+mins%3B%0A++++private+final+int+secs%3B%0A++++//rep+invariant%3A%0A++++//++++mins+%(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+main(String%5B%5D+args%29+%7B%1(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+++++System.out.println(%22d1.equals(o2%2frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLst.JSON=%5B%5D&curlInstr=49) in the Online Python Tutor.</p>

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.equals(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:

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

```
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?

READING EXERCISES

Equals-ish

Consider the latest implementation of `Duration` in the reading, reprinted here for convenience:

```
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;
    }

    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;
    }
}
```

Suppose these `Duration` objects are created:

```
Duration d_0_60 = new Duration(0, 60);
Duration d_1_00 = new Duration(1, 0);
Duration d_0_57 = new Duration(0, 57);
Duration d_1_03 = new Duration(1, 3);
```

Which of the following expressions return true?

- `d_0_60.equals(d_1_00)`
- `d_1_00.equals(d_0_60)`
- `d_1_00.equals(d_1_00)`

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

- `d_0_57.equals(d_1_00)`
- `d_0_57.equals(d_1_03)`
- `d_0_60.equals(d_1_03)`

➤ The `equals` method compares the total lengths of the intervals in seconds, and allows them to differ by up to 5 seconds (`CLOCK_SKEW`) while still testing equal. So all the objects are equal to each other except for `d_0_57` and `d_1_03`, which differ by 6 seconds.

CHECK

EXPLAIN

Skewed up

Which properties of an equivalence relation are violated by this `equals()` method? (Ignore null references.)

- `recursivity`
- `reflexivity`
- `sensitivity`
- `symmetry`
- `transitivity`

➤ This `equals()` violates transitivity: `d_0_57` equals `d_1_00`, and `d_1_00` equals `d_1_03`, but `d_0_57` does not equal `d_1_03`.

CHECK

EXPLAIN

Buggy equality

Suppose you want to show that an equality operation is buggy because it isn't reflexive. How many objects do you need for a counterexample to reflexivity?

- `none`
- `1 object`
- `2 objects`
- `3 objects`
- `all the objects in the type`

➤ If you show that `x.equals(x)` returns `false` for some particular object `x`, then you have a counterexample to reflexivity.

CHECK

EXPLAIN

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.

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

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 matches 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 (<http://stackoverflow.com/questions/113511/hash-code-implementation>). Recent versions of Java now have a utility method `Objects.hash()` (<http://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`!

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 EXERCISES

Give me the code

Consider the following ADT class:

```
class Person {  
    private String firstName;  
    private String lastName;  
    ...  
  
    public boolean equals(Object obj) {  
        if (!(obj instanceof Person)) return false;  
        Person that = (Person) obj;  
        return this.lastName.toUpperCase().equals(that.lastName.toUpperCase());  
    }  
  
    public int hashCode() {  
        // TODO  
    }  
}
```

Which of the following could be put in place of the line marked `TODO` to make `hashCode()` consistent with `equals()`?

- return 42;
- return `firstName.toUpperCase();`
- return `lastName.toUpperCase().hashCode();`
- return `firstName.hashCode() + lastName.hashCode();`

➤ The crucial property is that when two objects `o1` and `o2` are `equal()`, then `o1.hashCode()` must return the same value as `o2.hashCode()`.

42 trivially satisfies that property.

`firstName.toUpperCase()` is a `String`, not an `int`, so this line wouldn't even compile.

`lastName.toUpperCase().hashCode()` is correct, because it relies on the consistency of `String`'s `equals()` and `hashCode()` functions.

`firstName.hashCode() + lastName.hashCode()` wouldn't work for two reasons. First, two `Person` objects can have different `firstName` values and still compare equal, since only `lastName` is examined in `equals()`, but the different `firstName` values would lead to different hashcodes. Second, two `Person` objects whose last names differ in case, such as "FooBar" and "Foobar", would compare `equal()`, but likely have different hash codes.

CHECK

EXPLAIN

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.

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

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

([http://www.pythontutor.com/java.html#code=import+java.util.*%3B%0Apublic+class+WhyObservationalEqualityHurts+%7B%0A++public+static+void+main\(String%5B%5D+args%29+%7B%0A++++List%\(%3CString%3E+l+\)%3A+set%29+%7B%0A++++System.out.println\(%22set.contains\(%29%3D%22+%2B+set.contains\(%29%29%3B%0A++++%7D%0A++%7D%0A%7D&mode=display&origin=frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=13](http://www.pythontutor.com/java.html#code=import+java.util.*%3B%0Apublic+class+WhyObservationalEqualityHurts+%7B%0A++public+static+void+main(String%5B%5D+args%29+%7B%0A++++List%(%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++%7D%0A%7D&mode=display&origin=frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=13))

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.

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

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.

READING EXERCISES

Bag

Suppose `Bag<E>` is a mutable ADT representing what is often called a *multiset*, an unordered collection of objects where an object can occur more than once. It has the following operations:

```
/** make an empty bag */
public Bag<E>()

/** modify this bag by adding an occurrence of e, and return this bag */
public Bag<E> add(E e)

/** modify this bag by removing an occurrence of e (if any), and return this bag */
public Bag<E> remove(E e)

/** return number of times e occurs in this bag */
public int count(E e)
```

Suppose we run this code:

```
Bag<String> b1 = new Bag<String>().add("a").add("b");
Bag<String> b2 = new Bag<String>().add("a").add("b");
Bag<String> b3 = b1.remove("b");
Bag<String> b4 = new Bag<String>().add("b").add("a"); // swap!
```

You are not
logged in.

Which of the following expressions are true after all the the code has been run?

- ✗ `b1.count("a") == 1`
- `b1.count("b") == 1`
- `b2.count("a") == 1`
- `b2.count("b") == 1`
- `b3.count("a") == 1`
- `b3.count("b") == 1`
- `b4.count("a") == 1`

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

`b4.count("b") == 1`

➤ `b1` and `b3` are aliases for the same Bag, which ends up containing just one occurrence of "a" and none of "b".

`b2` and `b4` are both references to different Bags, each of which has one occurrence of "a" and one of "b".

CHECK

EXPLAIN

Bag behavior

If `Bag` is implemented with behavioral equality, which of the following expressions are true?

`b1.equals(b2)`

`b1.equals(b3)`

`b1.equals(b4)`

`b2.equals(b3)`

`b2.equals(b4)`

`b3.equals(b1)`

➤ Behavioral equality of mutable ADTs requires two references to be equal if and only if they are aliases for the same object. So `b1` and `b3` must compare equal, but `b2` must not be equal to them, and it must also not be equal to `b4`. Symmetry should also still apply.

CHECK

EXPLAIN

Bean bag

If `Bag` were part of the Java API, it would probably implement observational equality, counter to the recommendation in the reading.

If `Bag` implemented observational equality despite the dangers, which of the following expressions are true?

`b1.equals(b2)`

`b1.equals(b3)`

`b1.equals(b4)`

`b2.equals(b3)`

`b2.equals(b4)`

`b3.equals(b1)`

➤ Equality is now defined by the observer operation `count`, so `b1` and `b3` are certainly equal, but `b2` and `b4` are now considered equal as well.

The Java Collections implement observational equality because it is often convenient, but it would be better to implement a *different* operation for observational equality of mutable types.

CHECK

EXPLAIN

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:

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

```
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

([http://www.pythontutor.com/java.html#code=import+java.util.*%3B%0Apublic+class+AutounboxingProblem+%7B%0A++public+static+void+main\(String%5B%5D+args%29+%7B%0A++++Map%3CStrin\(a.get\(%22c%22%29+%3D%3D+b.get\(%22c%22%29%29+%29%3B%0A++%7D%0A%7D&mode=display&origin=opt-frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curlInstr=6\)](http://www.pythontutor.com/java.html#code=import+java.util.*%3B%0Apublic+class+AutounboxingProblem+%7B%0A++public+static+void+main(String%5B%5D+args%29+%7B%0A++++Map%3CStrin(a.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.

Collaboratively authored with contributions from: Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Gutttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. This work is licensed under CC BY-SA 4.0 (<http://creativecommons.org/licenses/by-sa/4.0/>).

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:

Collaboratively authored
with contributions from:
Saman Amarasinghe, Adam
Chlipala, Srinivas Devadas,
Reading 16: Recursive
Michael Ernst, Max
Data Types
John Guttag,
Daniel Jackson, Rob Miller,
Martin Rinard and
Armando Solar-Lezama.

This work is licensed under
Part 1: Recursive Data
CC BY-SA 4.0

Types (creativecommons.org/licenses/by-sa/4.0/).

Part 2: Writing a Program with ADTs

Summary

- **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.

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.

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

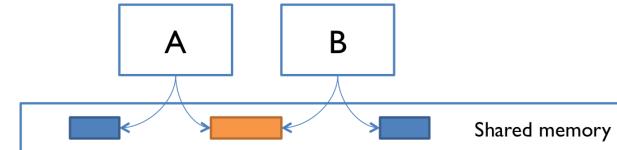
Summary

Two Models for Concurrent Programming

There are two common models for concurrent programming: *shared memory* and *message passing*.

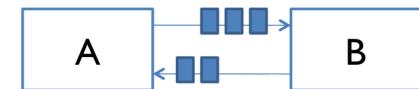
Shared memory. In the shared memory model of concurrency, concurrent modules interact by reading and writing shared objects in memory.

Other 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.

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

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.

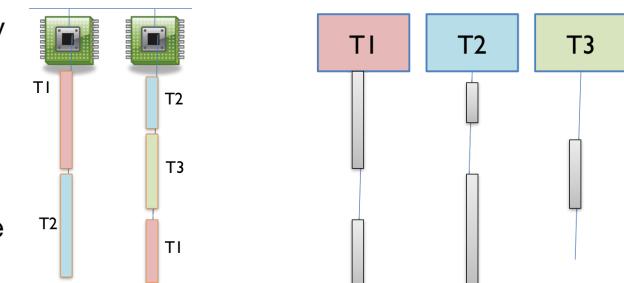
Thread. A thread is a locus of control inside a running program. Think of it as a place in the program that is being run, plus the stack of method calls that led to that place (so the thread can go back up the stack when it reaches `return` statements).

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:

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

- **Processes & Threads**
(<http://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html>) (just 1 page)
- **Defining and Starting a Thread**
(<http://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html>) (just 1 page)

The second Java Tutorials reading shows two ways to create a thread.

- Never use their second way (subclassed Thread).
- Always implement the Runnable (<http://docs.oracle.com/javase/8/docs/api/?java/lang/Runnable.html>) interface and use the new Thread(...) constructor.

READING EXERCISES

Processes and threads

When you run a Java program (for example, using the Run button in Eclipse), how many processors, processes, and threads are created at first?

Processors:



one processor



no processors

Processes:



one process



one process

Threads:



one thread



one thread



Processors are physical components in the machine. For our purposes, the number of processors is fixed.

When a program starts, it automatically gets one process to run in, and it gets one thread for its execution.

CHECK**EXPLAIN**

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

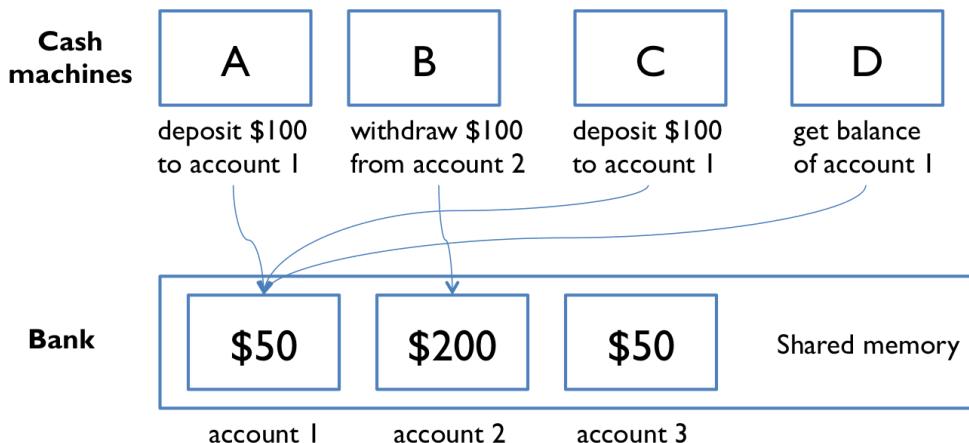
Concurrency is Hard
to Test and Debug

Summary

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
```

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

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):

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

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 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."

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

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

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

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**

(<http://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:

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

```
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`:

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

```
private void computeAnswer() {  
    boolean tmpr = ready;  
    int tmpa = answer;  
  
    tmpa = 42;  
    tmpr = true;  
  
    ready = tmpr;  
        // <-- what happens if useAnswer() interleaves here?  
        // ready is set, but answer isn't.  
    answer = tmpa;  
}
```

READING EXERCISES

Race conditions 1

Consider the following code:

```
private static int x = 1;  
  
public static void methodA() {  
    x *= 2;  
    x *= 3;  
}  
  
public static void methodB() {  
    x *= 5;  
}
```

Suppose `methodA` and `methodB` run **sequentially**, i.e. first one and then the other. What is the final value of `x`?



s

30

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

► If `methodA` runs first, then it sets `x` to $1 \times 2 \times 3 = 6$, and then `methodB` runs and sets `x` to $5 \times 6 = 30$. Since multiplication is commutative, we get the same result if `methodB` runs before `methodA`.

CHECK

EXPLAIN

Race conditions 2

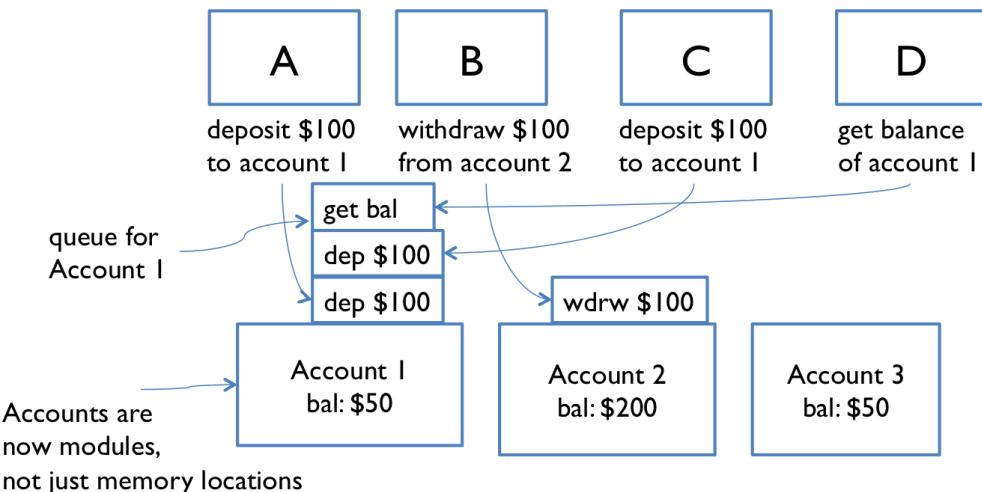
Message Passing Example

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

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

overdrawing 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.

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

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.

READING EXERCISES

Testing concurrency

You are not logged in.

You're running a JUnit test suite (for code written by somebody else), and some of the tests are failing. You add `System.out.println` statements to the one method called by all the failing test cases, in order to display some of its local variables, and the test cases suddenly start passing. Which of the following are likely reasons for this?

- The method is calling a random number generator (e.g. `Math.random()`), so sometimes its tests will pass by random chance.
 - The method has code running concurrently.
 - The method has a race condition.
 - The method's preconditions are not being met by the test cases.
- Randomness is not a likely reason for this behavior. Although calling a random number generator does make a method nondeterministic, which is one reason why tests might sometimes pass and sometimes fail, it's less likely that the original developer would have written tests that didn't take this into account, and still more unlikely that multiple tests would start to pass coincidentally with inserting a print statement.
- Concurrent code, which contains a race condition affecting its correctness, is likely to be affected by the timing changes caused by a print statement.
- If the method's preconditions are not being met by the test cases, this is more likely to cause deterministic failures (test cases that always fail), rather than nondeterministic behavior.

CHECK

EXPLAIN

Summary

Reading 19:
Concurrency

Concurrency

Two Models for
Concurrent
Programming

Collaboratively authored
[Processes, Threads, and Time Slicing](#)
with contributions from:
Adam Chlipala, Srini Devadas, Michael Ernst,
Max Goldman, John Guttag, Daniel Jackson,
Rob Miller, Martin Herwig, and Mando Solar-Lezama. This work
is licensed under CC BY-SA 4.0.
(<http://creativecommons.org/licenses/by-sa/4.0/>).
Tweaking the Code Won't Help

Reordering

Message Passing
Example

Concurrency is Hard
to Test and Debug

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.

MIT EECS accessibility (<https://accessibility.mit.edu>)

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
Strategy 1: Confinement	Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.
Strategy 2: Immutability			
Strategy 3: Using Threadsafe Data Types	<h3>Objectives</h3> <p>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.</p>		
How to Make a Safety Argument	<p>There are basically four ways to make variable access safe in shared-memory concurrency:</p> <ul style="list-style-type: none">• 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.		
Summary	<p>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.</p> <p>The material in this reading is inspired by an excellent book: Brian Goetz et al., <i>Java Concurrency in Practice</i> (http://jcip.net/), Addison-Wesley, 2006.</p>		

What Threadsafe Means

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

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` (<http://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:

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

```
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);  
    }  
}
```

Starting a thread with an **anonymous** (<https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html>) **Runnable** is a common idiom. Read:

- A quick explanation of **using an anonymous Runnable to start a thread (anonymous-runnable/)**.

Let's look at snapshot diagrams for this code. Hover or tap on each step to update the diagram:

Reading 20: Thread Safety

What Threadsafe Means

Strategy 1: Confinement

Strategy 2: Immutability

Strategy 3: Using Threadsafe Data Types

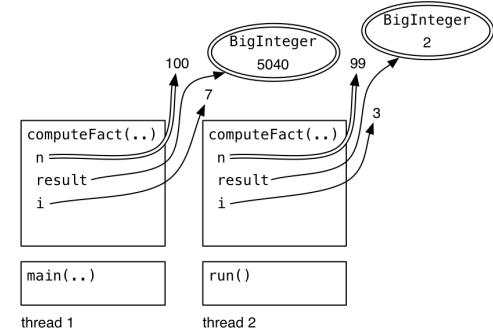
How to Make a Safety Argument

Summary

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:

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

```
// 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:

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

```
// is this method threadsafe?  
/**  
 * @param x integer to test for primeness; requires x > 1  
 * @return true if and only if x is prime  
 */  
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 (<http://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 (http://web.mit.edu/6.005/www/fa15/classes/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 (http://web.mit.edu/6.005/www/fa15/classes/19-concurrency/#shared_memory_example).

READING EXERCISES

Factorial

In the factorial example above, `main` looks like:

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

```
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);  
}
```

Which of the following are possible interleavings?

- ✗ The call to `computeFact(100)` starts before the call to `computeFact(99)` starts
- The call to `computeFact(99)` starts before the call to `computeFact(100)` starts
- The call to `computeFact(100)` finishes before the call to `computeFact(99)` starts
- The call to `computeFact(99)` finishes before the call to `computeFact(100)` starts
- All of these are possible.

CHECK

EXPLAIN

PinballSimulator

Here's part of the pinball simulator example above:

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

```
public class PinballSimulator {  
  
    private static PinballSimulator simulator = null;  
  
    // ...  
  
    public static PinballSimulator getInstance() {  
        1) if (simulator == null) {  
        2)     simulator = new PinballSimulator();  
        3)     return simulator;  
    }  
}
```

The code has a race condition that invalidates the invariant that only one simulator object is created.

Suppose two threads are running `getInstance()`. One thread is about to execute one of the numbered lines above; the other thread is about to execute the other. For each pair of possible line numbers, is it possible the invariant will be violated?

About to execute lines 1 and 3

- ✓ Yes, it could be violated ✕
 No, we're safe

➤ The thread on line 3 has already assigned `simulator`, so the thread on line 1 will not enter the conditional. Right?

Unfortunately, that's not correct. As we saw in the last reading (<http://web.mit.edu/6.005/www/fa15/classes/19-concurrency/#reordering>), Java doesn't guarantee that the assignment to `simulator` in one thread will be immediately visible in other threads; it might be cached temporarily. In fact, our reasoning is broken, and the invariant can still be violated.

About to execute lines 1 and 2

- ✗ Yes, it could be violated ✕
 No, we're safe

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

- If the thread about to execute line 1 goes first, both threads are inside the conditional and will create new simulator objects.

About to execute lines 1 and 1

- ✗ Yes, it could be violated
- No, we're safe

- If both threads test the predicate before either thread assigns `simulator`, both will enter the conditional and create new simulator objects.

CHECK

EXPLAIN

Confinement

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 (http://web.mit.edu/6.005/www/fa15/classes/16-recursive-data-types/recursiive/#tuning_the_rep) the first time the length was requested by a client. Caching is a typical kind of beneficent mutation.

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

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 (<http://web.mit.edu/6.005/www/fa15/classes/13-abstraction-functions-rep-invariants/#invariants>)
- no mutation whatsoever of mutable objects in the rep – not even beneficent mutation (http://web.mit.edu/6.005/www/fa15/classes/16-recursive-data-types/recursion/#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**
(<http://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html>) (1 page)

READING EXERCISES

Immutability

Suppose you're reviewing an abstract data type which is specified to be immutable, to decide whether its implementation actually is immutable and threadsafe.

Which of the following elements would you have to look at?

- fields
- creator implementations
- client calls to creators
- producer implementations

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

- client calls to producers
- observer implementations 
- client calls to observers
- mutator implementations
- client calls to mutators

► Fields need to be examined to make sure they're private and final.

Creator and producer implementations need to be checked for rep exposure – e.g., to make sure that a reference to a mutable object passed in from a client isn't being stored in the rep.

Observer implementations need to be checked for rep exposure as well, in this case returning a reference to a mutable object in the rep.

Client calls to operations do not need to be examined, because the ADT must guarantee its own immutability, regardless of what clients do.

Mutators do not need to be examined, because an ostensibly-immutable ADT shouldn't have any mutators.

CHECK

EXPLAIN

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` (<http://docs.oracle.com/javase/8/docs/api/java/lang/StringBuffer.html>) says:

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

[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 (<http://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.

Threadsafe 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.

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

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-threadsafe 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:

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

```
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**
(<http://docs.oracle.com/javase/tutorial/collections/implementations/wrapper.html>) (1 page)
- **Concurrent Collections**
(<http://docs.oracle.com/javase/tutorial/essential/concurrency/collections.html>) (1 page)

READING EXERCISES

Threading data types

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

Consider this class's rep:

```
public class Building {  
    private final String buildingName;  
    private int numberofFloors;  
    private final int[] occupancyPerFloor;  
    private final List<String> companyNames = Collections.synchronizedLi  
st(new ArrayList<>());  
    ...  
}
```

Which of these variables use a threadsafe data type?

- buildingName
- numberofFloors
- occupancyPerFloor
- companyNames

- String and int are both immutable, so the types themselves are threadsafe.
int[] is mutable, and not threadsafe.
- List<String> is not automatically threadsafe, but the implementation of List<String> used here is a synchronized list wrapper, and companyNames is final so it can never be assigned to a different List, so this type is threadsafe.

Which of these variables are safe for use by multiple threads?

- buildingName
- numberofFloors
- occupancyPerFloor
- companyNames

- Not only does the variable's type have to be threadsafe, but the variable itself should be final. buildingName and companyNames satisfy that, but reads and writes of numberofFloors may have race conditions.

Which of these variables cannot be involved in any race condition?

- buildingName

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

- ✖ numberOfFloors
- occupancyPerFloor
- companyNames

➤ buildingName is immutable, so it can't be involved in any race condition.

companyNames might still be involved in a race condition caused by (otherwise safe) mutations to the list, e.g.:

```
if (companyNames.size() > 0) { String firstCompany = companyNames.get(0); }
```

If another thread could empty the companyNames list between the size check and the get call, then this code will fail.

CHECK

EXPLAIN

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.

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

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:

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

```
/** 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
    //     String 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:

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

```
/** StringBuffer is a threadsafe mutable string of characters. */
public class StringBuffer {
    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

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

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.

READING EXERCISES

Safety arguments

Consider the following ADT with a **bad** safety argument that appeared above:

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

```
/** StringBuffer is a threadsafe mutable string of characters. */
public class StringBuffer {
    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

    /** @return the string represented by this buffer,
     *          with all letters converted to uppercase */
    public String toUpperCase() { return text.toUpperCase(); }

    /** @param pos position to insert text into the buffer,
     *             requires 0 <= pos <= length of the current string
     * @param s text to insert
     * Mutates this buffer to insert s as a substring at position pos.
    */
    public void insert(int pos, String s) {
        text = text.substring(0, pos) + s + text.substring(pos);
    }

    /** @return the string represented by this buffer */
    public void toString() { return text; }

    /** Resets this buffer to the empty string. */
    public void clear() { text = ""; }

    /** @return the first character of this buffer, or "" if this buffer
     * is empty */
    public String first() {
        if (text.length() > 0) {
            return String.valueOf(text.charAt(0));
        } else {
            return "";
        }
    }
}
```

```
}
```

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

Which of these methods are counterexamples to the buggy safety argument, because they have a race condition?

In particular, you should mark method A as a counterexample if it's possible that, if one thread is running method A at the same time as another thread is running some other method, some interleaving would violate A's postcondition:

- ✖ toUpperCase
- insert ✅
- toString
- clear
- first ✅

➤ Both insert and first may fail if clear interleaves at the wrong moment – they will throw IndexOutOfBoundsException exceptions, which are not permitted by their specs, assuming their precondition was satisfied when the method started.

toUpperCase, toString, and clear will not violate their own postconditions. toUpperCase and toString are just observers, in fact, so they won't be able to hurt other methods by interleaving with them.

CHECK

EXPLAIN

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 StringBuffer sb representing "z". They

A

B

call sb.clear()

call sb.insert(0, "a")

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

run `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.

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

`call sb.clear()`

— in `clear`: `text = ""`

— `clear` returns

B

— in `clear`: `text = ""`

— in `insert`: `text = "" + "a" + "z"`

— `clear` returns

— `insert` returns

`assert sb.toString()
 .equals("")`

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

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

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.

READING EXERCISES

Serializability

Suppose two threads are sharing a `StringBuffer` representing "z".

For each pair of concurrent calls and their result, does that outcome violate serializability (and therefore demonstrate that `StringBuffer` is not threadsafe)?

`clear()` and `insert(0, "a")` → `insert` throws an `IndexOutOfBoundsException`

- ✖ Violates serializability
- Consistent with serializability

➤ Calling `insert` with `pos=0` should never violate its precondition, so throwing the exception is not consistent with any sequential ordering.

`clear()` and `insert(1, "a")` → `insert` throws an `IndexOutOfBoundsException`

- ✓ Violates serializability
- Consistent with serializability

➤ This is consistent with `clear`-followed-by-`insert`. The precondition `pos <= length` of the current string is violated, since `clear` happened first.

`first()` and `insert(0, "a")` → `first` returns "a"

- ✓ Violates serializability
- Consistent with serializability

➤ This is consistent with `insert`-followed-by-`first`.

`first()` and `clear()` → `first` returns "z"

- ✖ Violates serializability

You are not logged in.

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

Collaboratively authored with contributions from: Saman Amarasinghe, Adam Chlipala, Srinivas Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. This work is licensed under CC BY-SA 4.0 (<http://creativecommons.org/licenses/by-sa/4.0/>).

- Consistent with serializability
- This is consistent with `first`-followed-by-`clear`.
`first()` and `clear()` → `first` throws an `IndexOutOfBoundsException`
- Violates serializability
 Consistent with serializability
- The exception is never allowed by `first`'s postcondition, so throwing it is not consistent with any sequential ordering.

CHECK

EXPLAIN

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.

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

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

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.

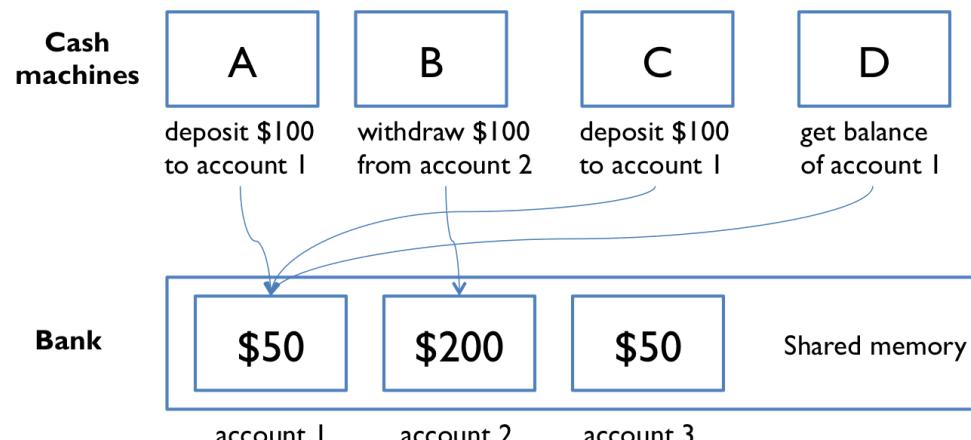
Using a lock also tells the compiler and processor that you’re using shared memory concurrently, so that registers and caches will be flushed out to shared storage. This avoids the problem of reordering (./19-concurrency/#reordering), ensuring that the owner of a lock is always looking at up-to-date 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).

Reading 23: Locks and Synchronization

Introduction

Synchronization

Deadlock

Developing a threadsafe abstract data type

Locking

Monitor pattern

Thread safety argument with synchronization

Atomic operations

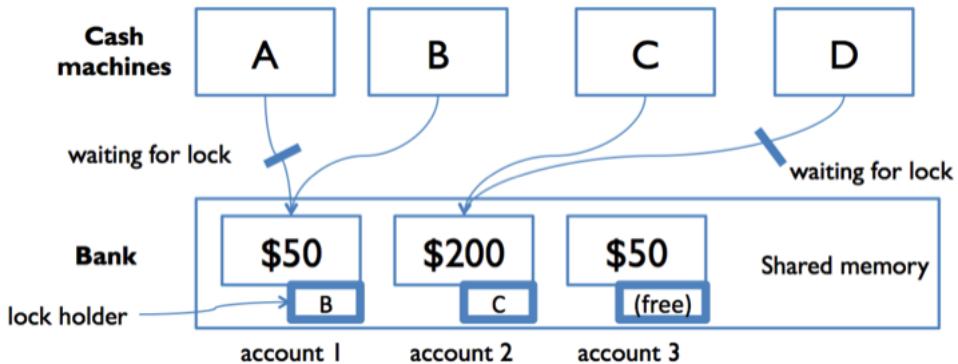
Designing a datatype for concurrency

Deadlock rears its ugly head

Goals of concurrent program design

Concurrency in practice

Summary



Deadlock

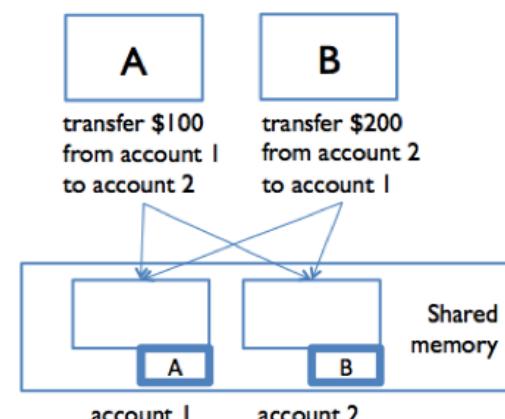
When used properly and carefully, locks can prevent race conditions. But then another problem rears its ugly head. Because the use of locks requires threads to wait (acquire blocks when another thread is holding the lock), it's possible to get into a situation where two threads are waiting for each other — 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.



- **Deadlock** (<http://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html>) (1 page)

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

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/fa15-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:

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

```
/** An EditBuffer represents a threadsafe mutable string of characters
 * in a text editor. */
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:

```
public class SimpleBuffer implements EditBuffer {
    private String text;
    // Rep invariant:
    //   text != null
    // Abstraction function:
    //   represents the sequence text[0],...,text[text.length()-1]
```

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

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 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:

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

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 `Objects` are often used for explicit locking:

```
Object lock = new Object();
```

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

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)`.

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

```
/** 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**
(<http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html>) (1 page)
- **Intrinsic Locks and Synchronization**
(<http://docs.oracle.com/javase/tutorial/essential/concurrency/locksSync.html>) (1 page)

READING EXERCISES

Synchronizing with locks

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

If thread B tries to acquire a lock currently held by thread A:

What happens to thread A?

- ✗ blocks until B acquires the lock
✗ blocks until B releases the lock
✗ nothing ✓

What happens to thread B?

- ✗ blocks until A acquires the lock
✗ blocks until A releases the lock ✓
✗ nothing

➤ Thread B blocks until A releases the lock. Then B acquires the lock.

CHECK

EXPLAIN

This list is mine, all mine

Suppose `list` is an instance of `ArrayList<String>`.

What is true while A is in a `synchronized (list) { ... }` block?

- ✗ it owns the lock on `list` ✓
✗ it does not own the lock on `list`
✗ no other thread can use observers of `list`
✗ no other thread can use mutators of `list`
✗ no other thread can acquire the lock on `list` ✓
✗ no other thread can acquire locks on elements in `list`

➤ Acquiring a lock means one thing and one thing only: no other thread can acquire that lock, until the lock is released.

The spec of `ArrayList` could advertise that acquiring the lock on an instance of `ArrayList` provides some synchronization benefit, but it does not.

Or, if you wanted acquiring the lock on `list` to mean some of the other things above, all the threads would have to agree on that locking discipline.

CHECK

EXPLAIN

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

OK fine but this synchronized List is totally mine

Suppose `sharedList` is a `List` returned by `Collections.synchronizedList` (<http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#synchronizedList--java.util.List->).

It is now safe to use `sharedList` from multiple threads without acquiring any locks... except! Which of the following would require a `synchronized(sharedList) { ... }` block?

- ✖ call `isEmpty`
 call `add`
 iterate over the list
 call `isEmpty`, if it returns false, call `remove(0)`

➤ The `sharedList` is safe for concurrency, so individual operations are safe to call without additional synchronization.

The spec of `Collections.synchronizedList` explicitly says that you must synchronize on the list before iterating. This prevents other clients from mutating the list during the iteration.

We must also synchronize on the list any time we need to maintain an invariant between calls to individual operations: in between our call to `isEmpty` and `remove`, someone else could have emptied the list!

CHECK

EXPLAIN

I heard you like locks so I acquired your lock so you can lock while you acquire

Suppose we run this code:

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

```
synchronized (obj) {  
    // ...  
    synchronized (obj) { // <-- uh oh, deadlock?  
        // ...  
    }  
    // <-- do we own the lock on obj?  
}
```

On the line “uh oh, deadlock?”, do we experience deadlock?

- yes
- no

If we don’t deadlock, on the line “do we own the lock on obj”, does the thread own the lock on obj?

- yes
- no
- we deadlocked

➤ A thread is allowed to re-acquire a lock it already owns (the technical term for this is *reentrant locks*).

Acquire and release come in pairs, and `synchronized` blocks on the same object can be safely nested inside each other. This means that a lock actually stores a *counter* of the number of times that its owner has acquired it without yet releasing it. The thread continues to own it until each acquire has had its corresponding release, and the counter has fallen to zero. So on the “do we own the lock” line, the thread does still have a lock on `obj`.

CHECK

EXPLAIN

Thread safety argument with synchronization

Now that we’re protecting `SimpleBuffer`’s `rep` with a lock, we can write a better thread safety argument:

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

```
/** 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:

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

```
/** 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`:

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

```
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, ...){
```

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

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` (<http://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

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

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);  
        }  
    }  
}
```

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

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.

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

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.

READING EXERCISES

Deadlock

You are not
logged in.

In the code below three threads 1, 2, and 3 are trying to acquire locks on objects `alpha`, `beta`, and `gamma`.

Thread 1

Thread 2

Thread 3

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

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

```
synchronized (alpha) {  
    // using alpha  
    // ...  
}  
  
synchronized (gamma) {  
    synchronized (beta) {  
        // using alpha, beta, & gamma  
        // ...  
    }  
}  
// finished
```

```
synchronized (gamma) {  
    synchronized (alpha) {  
        synchronized (beta) {  
            // using alpha, beta, & gamma  
            // ...  
        }  
    }  
}  
// finished
```

```
synchronized (gamma) {  
    synchronized (alpha) {  
        // using alpha & gamma  
        // ...  
    }  
}  
  
synchronized (beta) {  
    synchronized (gamma) {  
        // using beta & gamma  
        // ...  
    }  
}  
// finished
```

This system is susceptible to deadlock.

For each of the scenarios below, determine whether the system is in deadlock if the threads are currently on the indicated lines of code.

Scenario A

Thread 1 inside using alpha

Thread 2 blocked on synchronized (alpha)

Thread 3 finished

deadlock

not deadlock

➤ Thread 1 will exit the top synchronized block, release the lock on alpha , and the system will continue.

Scenario B

Thread 1 finished

Thread 2 blocked on synchronized (beta)

Thread 3 blocked on 2nd synchronized (gamma)

deadlock

not deadlock

- Thread 2 has acquired the lock on gamma and is awaiting beta . Thread 3 has beta and wants gamma . Deadlock.

Scenario C

Thread 1 running synchronized (beta)
Thread 2 blocked on synchronized (gamma)
Thread 3 blocked on 1st synchronized (gamma)

- ✓ deadlock
 not deadlock

- Thread 1 can successfully acquire the lock on beta , then exit the synchronized block, and one of the other threads will be able to acquire the lock on gamma . (As we saw in scenario B, they could deadlock later!)

Scenario D

Thread 1 blocked on synchronized (beta)
Thread 2 finished
Thread 3 blocked on 2nd synchronized (gamma)

- ✓ deadlock
 not deadlock

- Thread 1 has acquired the lock on gamma and is awaiting beta . Thread 3 has beta and wants gamma . Deadlock.

CHECK

EXPLAIN

Locked out

Examine the code again.

In the previous problem, we saw deadlocks involving beta and gamma .

What about alpha ?

- ✗ there is a possible deadlock where thread 1 owns the lock on alpha
 there is a possible deadlock where thread 2 owns the lock on alpha

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

there is a possible deadlock where thread 3 owns the lock on alpha

there are no deadlocks involving alpha

We can reason about it this way: in order to encounter deadlock, threads must try to acquire locks in different orders, creating a cycle in the graph of who-is-waiting-for-who.

So we look at alpha vs. beta : are there two threads that try to acquire these locks in the opposite order? No. Only thread 2 acquires them both at the same time.

Next we look at alpha vs. gamma : are there two threads that try to acquire these locks in the opposite order? No. Both thread 2 and thread 3 acquire both locks, but both of them acquire gamma first, then alpha .

Use the same analysis to demonstrate why beta and gamma are susceptible to deadlock: what order are these locks acquired in by the different threads?

CHECK

EXPLAIN

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?

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

Collaboratively authored with contributions from:

Saman Amarasinghe, Adam Summarayini Devadas,

Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama.

This work is licensed under CC BY-SA 4.0

(<http://creativecommons.org/licenses/by-sa/4.0/>).

- **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.

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