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

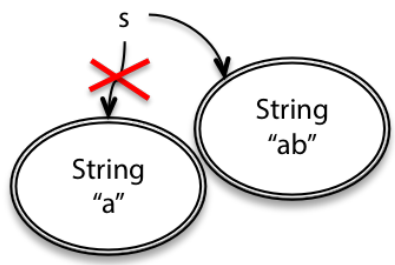
Mutability

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.

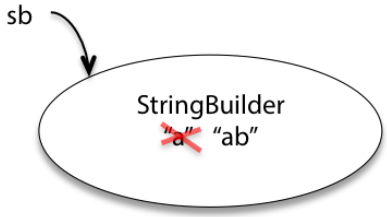
`String` (http://docs.oracle.com/javase/8/docs/api/?java/lang/String.html) is an example of an immutable type. A `String` object always represents the same string. `StringBuilder` (http://docs.oracle.com/javase/8/docs/api/?java/lang/StringBuilder.html) is an example of a mutable type. It has methods 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:

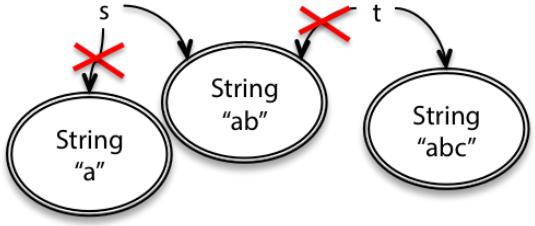


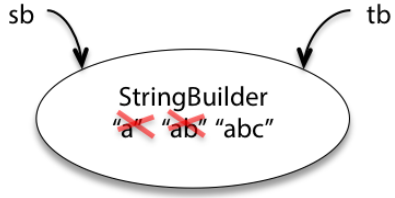

```
StringBuilder sb = new StringBuilder("a");
sb.append("b");
```

`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.

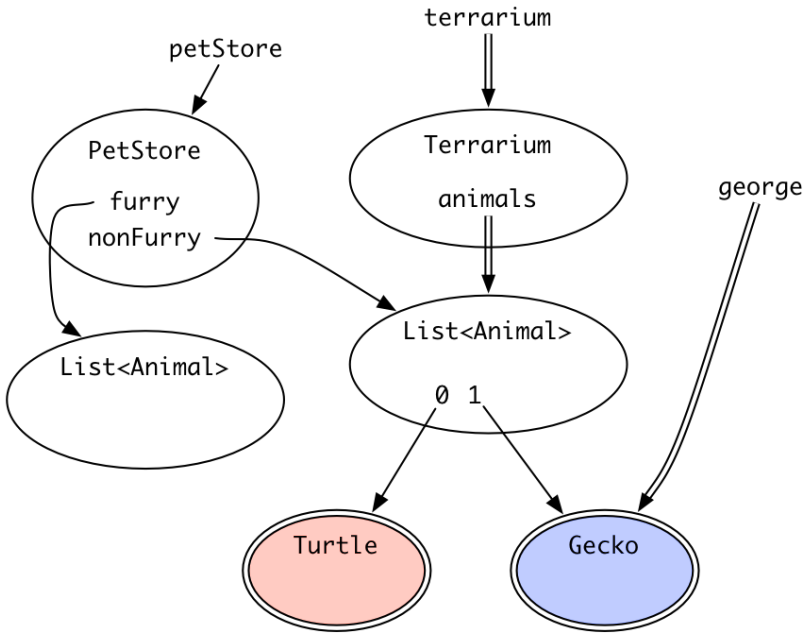
`StringBuilder` is designed to minimize this copying. It uses a simple but clever internal data structure to avoid doing any copying at all until the very end, when you ask for the final `String` with a `toString()` call:

```
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.

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

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:

```
// 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

✖ ⏥ 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?

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:

```
// 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?)

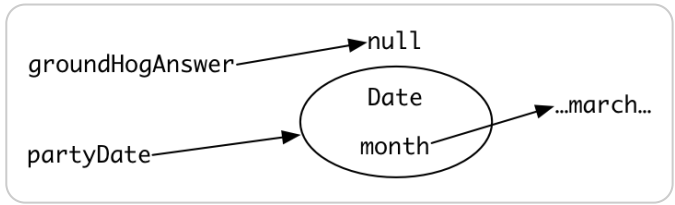
What happens when these two decisions interact? Even worse, think about who will first discover this bug — will it be `startOfSpring()` ? Will it be `partyPlanning()` ? Or will it be some completely innocent third piece of code that also calls `startOfSpring()` ?

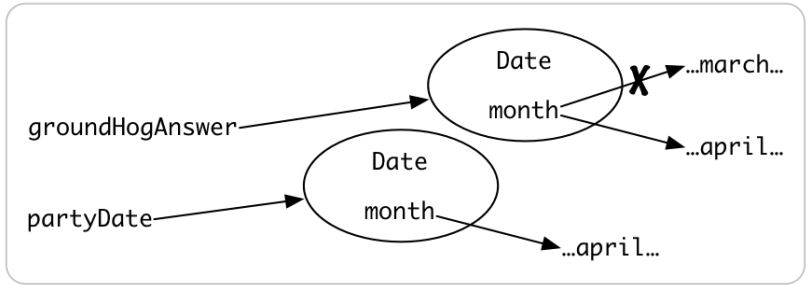
READING EXERCISES

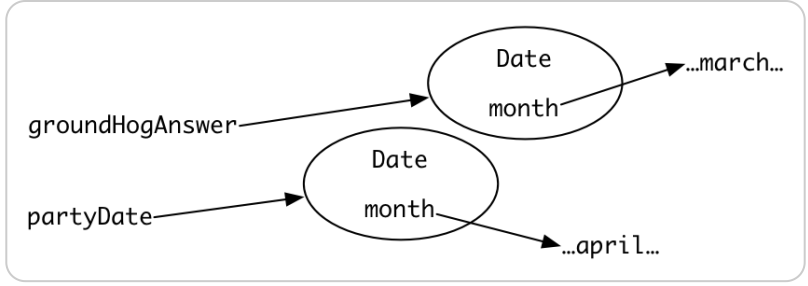
Risky #2

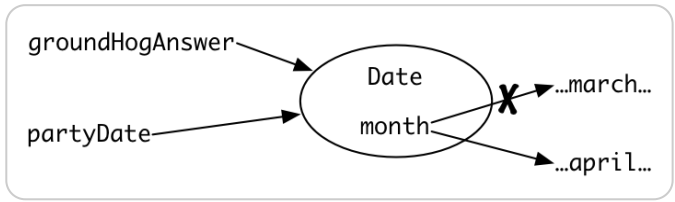
We don't know how `Date` stores the month, so we'll represent that with the abstract values `...march...` and `...april...` in an imagined `month` field of `Date` .

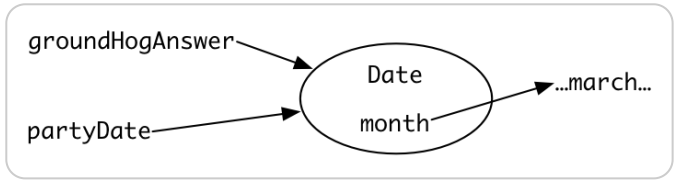
Which of these snapshot diagrams shows the bug?











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

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

> 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.

CHECK EXPLAIN

Key points:

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

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

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

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

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

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

Aliasing is what makes mutable types risky

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

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

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

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

Specifications for mutating methods

At this point it should be clear that when a method performs mutation, it is crucial to include that mutation in the method's spec, using the structure we discussed in the previous reading (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

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.

`MyIterator`

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

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

Instance variables (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`.

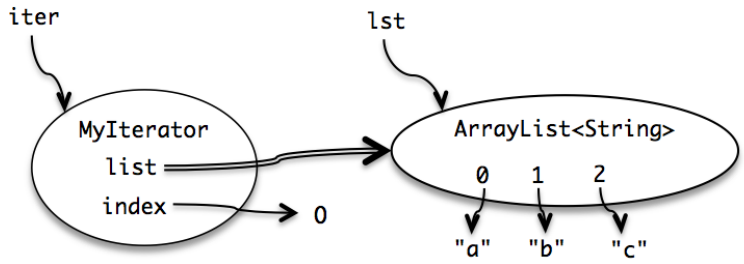
`private` is used for the object's internal state and internal helper methods, while `public` indicates methods and constructors that are intended for clients of the class (access control (http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html)).

`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).

```
/**
 * A MyIterator is a mutable object that iterates over
 * the elements of an ArrayList<String>, from first to last.
 * This is just an example to show how an iterator works.
 * In practice, you should use the ArrayList's own iterator
 * object, returned by its iterator() method.
 */
public class MyIterator {

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

    /**
     * Make an iterator.
     * @param list list to iterate over
     */
    public MyIterator(ArrayList<String> list) {
        this.list = list;
        this.index = 0;
    }

    /**
     * Test whether the iterator has more elements to return.
     * @return true if next() will return another element,
     *         false if all elements have been returned
     */
    public boolean hasNext() {
        return index < list.size();
    }

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

```
        return element;
    }
}
```


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



Note that we draw the arrow from `list` with a double line, to indicate that it's *final*. That means the arrow can't change once it's drawn. But the `ArrayList` object it points to is mutable — elements can be changed within it — and declaring `list` as final has no effect on that.

Why do iterators exist? There are many kinds of collection data structures (linked lists, maps, hash tables) with different kinds of internal representations. The iterator concept allows a single uniform way to access them all, so that client code is simpler and the collection implementation can change without changing the client code that iterates over it. Most modern languages (including Python, C#, and Ruby) use the notion of an iterator. It's an effective **design pattern** (a well-tested solution to a common design problem). We'll see many other design patterns as we move through the course.

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:

```
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*
- [x] `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*
- [x] `ArrayList`
- [ ] `MyIterator`

- [ ] `String` ☑
- [ ] `boolean`
- [ ] `int`

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

CHECK EXPLAIN

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`
- [x] `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*
- [x] `this`

- [ ] `hasNext`
- [ ] the return value ☑

> 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*
- [x] `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:

```
// 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:

```
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.

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

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

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

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

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

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

Here's a spec with a similar problem:

```
public static char[] 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.

- 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

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
- [x] 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.

> 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 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.

```
    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.
- [x] The reference to `that.amount` is allowed by Java. ☑

- [ ] 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.
- [x] 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();
```

- [ ] 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.
- [x] 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.
- [x] 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);
```

- [x] 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.
- [x] 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.
- [x] 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.
- [x] The illegal access is caught statically.
- [ ] The illegal access is caught dynamically.

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

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.
- [x] 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.
- [x] 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:

- **Abstraction.** Omitting or hiding low-level details with a simpler, higher-level idea.
- **Modularity.** Dividing a system into components or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system.
- **Encapsulation.** Building walls around a module (a hard shell or capsule) so that the module is responsible for its own internal behavior, and bugs in other parts of the system can't damage its integrity.
- **Information hiding.** Hiding details of a module's implementation from the rest of the system, so that those details can be changed later without changing the rest of the system.
- **Separation of concerns.** Making a feature (or "concern") the responsibility of a single module, rather than spreading it across multiple modules.

As a software engineer, you should know these terms, because you will run into them frequently. The fundamental purpose of all of these ideas is to help achieve the three important properties that we care about in 6.005: safety from bugs, ease of understanding, and readiness for change.

User-Defined Types

In the early days of computing, a programming language came with built-in types (such as integers, booleans, strings, etc.) and built-in procedures, e.g., for input and output. Users could define their own procedures: that's how large programs were built.

A major advance in software development was the idea of abstract types: that one could design a programming language to allow user-defined types, too. This idea came out of the work of many researchers, notably Dahl (the inventor of the Simula language), Hoare (who developed many of the techniques we now use to reason about abstract types), Parnas (who coined the term information hiding and first articulated the idea of organizing program modules around the secrets they encapsulated), and here at MIT, Barbara Liskov and John Guttag, who did seminal work in the specification of abstract types, and in programming language support for them – and developed the original 6.170, the predecessor to 6.005. Barbara Liskov earned the Turing Award, computer science's equivalent of the Nobel Prize, for her work on abstract types.

The key idea of data abstraction is that a type is characterized by the operations you can perform on it. A number is something you can add and multiply; a string is something you can concatenate and take substrings of; a boolean is something you can negate, and so on. In a sense, users could already define their own types in early programming languages: you could create a record type date, for example, with integer fields for day, month, and year. But what made abstract types new and different was the focus on operations: the user of the type would not need to worry about how its values were actually stored, in the same way that a programmer can ignore how the compiler actually stores integers. All that matters is the operations.

In Java, as in many modern programming languages, the separation between built-in types and user-defined types is a bit blurry. The classes in java.lang, such as Integer and Boolean are built-in; whether you regard all the collections of java.util as built-in is less clear (and not very important anyway). Java complicates the issue by having primitive types that are not objects. The set of these types, such as int and boolean, cannot be extended by the user.

Classifying Types and Operations

Types, whether built-in or user-defined, can be classified as **mutable** or **immutable**. The objects of a mutable type can be changed: that is, they provide operations which when executed cause the results of other operations on the same object to give different results. So `Date` is mutable, because you can call `setMonth` and observe the change with the `getMonth` operation. But `String` is immutable, because its operations create new `String` objects rather than changing existing ones. Sometimes a type will be provided in two forms, a mutable and an immutable form. `StringBuilder` , for example, is a mutable version of `String` (although the two are certainly not the same Java type, and are not interchangeable).

The operations of an abstract type are classified as follows:

- **Creators** create new objects of the type. A creator may take an object as an argument, but not an object of the type being constructed.
- **Producers** create new objects from old objects of the type. The concat method of `String` , for example, is a producer: it takes two strings and produces a new one representing their concatenation.
- **Observers** take objects of the abstract type and return objects of a different type. The `size` method of `List` , for example, returns an `int` .
- **Mutators** change objects. The `add` method of `List` , for example, mutates a list by adding an element to the end.

We can summarize these distinctions schematically like this (explanation to follow):

- creator: t* → T
- producer: T+, t* → T
- observer: T+, t* → t
- mutator: T+, t* → 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.

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 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.

Example: Different Representations for Strings

Let's look at a simple abstract data type to see what representation independence means and why it's useful. The `MyString` type below has far fewer operations than the real Java `String`, and their specs are a little different, but it's still illustrative. Here are the specs for the ADT:

```
/** MyString represents an immutable sequence of characters. */
public class MyString {

    ////////////////////// Example of a creator operation ///////////////
    /** @param b a boolean value
     *  @return string representation of b, either "true" or "false" */
    public static MyString valueOf(boolean b) { ... }

    ////////////////////// Examples of observer operations ///////////////
    /** @return number of characters in this string */
    public int length() { ... }

    /** @param i character position (requires 0 <= i < string length)
     *  @return character at position i */
    public char charAt(int i) { ... }

    ////////////////////// Example of a producer operation ///////////////
    /** Get the substring between start (inclusive) and end (exclusive).
     *  @param start starting index
     *  @param end ending index.  Requires 0 <= start <= end <= string length.
     *  @return string consisting of charAt(start)...charAt(end-1) */
    public MyString substring(int start, int end) { ... }
}
```

These public operations and their specifications are the only information that a client of this data type is allowed to know. Following the test-first programming paradigm, in fact, the first client we should create is a test suite that exercises these operations according to their specs. At the moment, however, writing test cases that use `assertEquals` directly on `MyString` objects wouldn't work, because we don't have an equality operation defined on `MyString`. We'll talk about how to implement equality carefully in a later reading. For now, the only operations we can perform with MyStrings are the ones we've defined above: `valueOf`, `length`, `charAt`, and `substring`. Our tests have to limit themselves to those operations. For example, here's one test for the `valueOf` operation:

```
MyString s = MyString.valueOf(true);
assertEquals(4, s.length());
assertEquals('t', s.charAt(0));
assertEquals('r', s.charAt(1));
assertEquals('u', s.charAt(2));
assertEquals('e', s.charAt(3));
```

We'll come back to the question of testing ADTs at the end of this reading.

For now, let's look at a simple representation for `MyString` : just an array of characters, exactly the length of the string, with no extra room at the end. Here's how that internal representation would be declared, as an instance variable within the class:

```
private char[] a;
```

With that choice of representation, the operations would be implemented in a straightforward way:

```
public static MyString valueOf(boolean b) {
    MyString s = new MyString();
    s.a = b ? new char[] { 't', 'r', 'u', 'e' }
            : new char[] { 'f', 'a', 'l', 's', 'e' };
    return s;
}

public int length() {
    return a.length;
}

public char charAt(int i) {
    return a[i];
}

public MyString substring(int start, int end) {
    MyString that = new MyString();
    that.a = new char[end - start];
    System.arraycopy(this.a, start, that.a, 0, end - start);
    return that;
}
```

Question to ponder: Why don't `charAt` and `substring` have to check whether their parameters are within the valid range? What do you think will happen if the client calls these implementations with illegal inputs?

One problem with this implementation is that it's passing up an opportunity for performance improvement. Because this data type is immutable, the `substring` operation doesn't really have to copy characters out into a fresh array. It could just point to the original `MyString` object's character array and keep track of the start and end that the new substring object represents. The String implementation in some versions of Java do this.

To implement this optimization, we could change the internal representation of this class to:

```
private char[] a;
private int start;
private int end;
```

With this new representation, the operations are now implemented like this:

```
    public static MyString valueOf(boolean b) {
        MyString s = new MyString();
        s.a = b ? new char[] { 't', 'r', 'u', 'e' }
                : new char[] { 'f', 'a', 'l', 's', 'e' };
        s.start = 0;
        s.end = s.a.length;
        return s;
    }

    public int length() {
        return end - start;
    }

    public char charAt(int i) {
      return a[start + i];
    }

    public MyString substring(int start, int end) {
        MyString that = new MyString();
        that.a = this.a;
        that.start = this.start + start;
        that.end = this.start + end;
        return that;
    }
```

Because `MyString` 's existing clients depend only on the specs of its public methods, not on its private fields, we can make this change without having to inspect and change all that client code. That's the power of representation independence.

READING EXERCISES

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 dupli
cates.
    public List<Person> people;

    /**
     * @return a list containing all the members of the family, with no duplic
ates.
     */
    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:

```
/**
 * 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 duplic
ates.
     */
    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

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()`.

CHECK EXPLAIN

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 satisifes its contract, even with the new `Set` representation, `client3` will keep working after the change. It is independent of the representation.

CHECK EXPLAIN

Representation 4

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 — specification

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

✔ specification — specification

```
public class Family {
```

✔ representation — representation

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

✖ implementation — representation

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

✖ implementation — specification

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

✖ representation — specification

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

✖ specification — implementation

```
        return people;
```

```
    }
}
```

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.toList()` (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	Instance method	`List.get()` (http://docs.oracle.com/javase/8/docs/api/java/util/List.html#get-int-)

		Static method	`Collections.max()` (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#max-java.util.Collection-)
	Producer operation	Instance method	`String.trim()` (http://docs.oracle.com/javase/8/docs/api/java/lang/String.html#trim--)
		Static method	`Collections.unmodifiableList()` (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#unmodifiableList-java.util.List-)
	Mutator operation	Instance method	`List.add()` (http://docs.oracle.com/javase/8/docs/api/java/util/List.html#add-E-)
		Static method	`Collections.copy()` (http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#copy-java.util.List-java.util.List-)
	Representation	`private` fields	

Testing an Abstract Data Type

We build a test suite for an abstract data type by creating tests for each of its operations. These tests inevitably interact with each other. The only way to test creators, producers, and mutators is by calling observers on the objects that result, and likewise, the only way to test observers is by creating objects for them to observe.

Here's how we might partition the input spaces of the four operations in our `MyString` type:

```
// testing strategy for each operation of MyString:
//
// valueOf():
//    true, false
// length():
//    string len = 0, 1, n
//    string = produced by valueOf(), produced by substring()
// charAt():
//    string len = 1, n
//    i = 0, middle, len-1
//    string = produced by valueOf(), produced by substring()
// substring():
//    string len = 0, 1, n
//    start = 0, middle, len
//    end = 0, middle, len
//    end-start = 0, n
//    string = produced by valueOf(), produced by substring()
```

Then a compact test suite that covers all these partitions might look like:

```
@Test public void testValueOfTrue() {
    MyString s = MyString.valueOf(true);
    assertEquals(4, s.length());
    assertEquals('t', s.charAt(0));
    assertEquals('r', s.charAt(1));
    assertEquals('u', s.charAt(2));
    assertEquals('e', s.charAt(3));
}

@Test public void testValueOfFalse() {
    MyString s = MyString.valueOf(false);
    assertEquals(5, s.length());
    assertEquals('f', s.charAt(0));
    assertEquals('a', s.charAt(1));
    assertEquals('l', s.charAt(2));
    assertEquals('s', s.charAt(3));
    assertEquals('e', s.charAt(4));
}

@Test public void testEndSubstring() {
    MyString s = MyString.valueOf(true).substring(2, 4);
    assertEquals(2, s.length());
    assertEquals('u', s.charAt(0));
    assertEquals('e', s.charAt(1));
}

@Test public void testMiddleSubstring() {
    MyString s = MyString.valueOf(false).substring(1, 2);
    assertEquals(1, s.length());
    assertEquals('a', s.charAt(0));
}

@Test public void testSubstringIsWholeString() {
    MyString s = MyString.valueOf(false).substring(0, 5);
    assertEquals(5, s.length());
    assertEquals('f', s.charAt(0));
    assertEquals('a', s.charAt(1));
    assertEquals('l', s.charAt(2));
    assertEquals('s', s.charAt(3));
    assertEquals('e', s.charAt(4));
}

@Test public void testSubstringOfEmptySubstring() {
```

```
    MyString s = MyString.valueOf(false).substring(1, 1).substring(0, 0);
    assertEquals(0, s.length());
}
```

Notice that each test case typically calls a few operations that *make* or *modify* objects of the type (creators, producers, mutators) and some operations that *inspect* objects of the type (observers). As a result, each test case covers parts of several operations.

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

How many parts are there in a reasonable, but not exhaustive, input-space partition of `promote()`?

✖ 1

How many parts are there in a reasonable, but not exhaustive, input-space partition of `getYears()`?

✖ 1

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

Software in 6.005

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

Objectives

Today's reading introduces several ideas:

- invariants
- representation exposure
- abstraction functions
- representation invariants

In this reading, we study a more formal mathematical idea of what it means for a class to implement an ADT, via the notions of *abstraction functions* and *rep invariants*. These mathematical notions are eminently practical in software design. The abstraction function will give us a way to cleanly define the equality operation on an abstract data type (which we'll discuss in more depth in a future class). The rep invariant will make it easier to catch bugs caused by a corrupted data structure.

Invariants

Resuming our discussion of what makes a good abstract data type, the final, and perhaps most important, property of a good abstract data type is that it **preserves its own invariants**. An *invariant* is a property of a program that is always true, for every possible runtime state of the program. Immutability is one crucial invariant that we've already encountered: once created, an immutable object

should always represent the same value, for its entire lifetime. Saying that the ADT *preserves its own invariants* means that the ADT is responsible for ensuring that its own invariants hold. It doesn't depend on good behavior from its clients.

When an ADT preserves its own invariants, reasoning about the code becomes much easier. If you can count on the fact that Strings never change, you can rule out that possibility when you're debugging code that uses Strings – or when you're trying to establish an invariant for another ADT that uses Strings. Contrast that with a string type that guarantees that it will be immutable only if its clients promise not to change it. Then you'd have to check all the places in the code where the string might be used.

Immutability

Later in this reading, we'll see many interesting invariants. Let's focus on immutability for now. Here's a specific example:

```
/**
 * This immutable data type represents a tweet from Twitter.
 */
public class Tweet {

    public String author;
    public String text;
    public Date timestamp;

    /**
     * Make a Tweet.
     * @param author    Twitter user who wrote the tweet.
     * @param text      text of the tweet
     * @param timestamp date/time when the tweet was sent
     */
    public Tweet(String author, String text, Date timestamp) {
        this.author = author;
        this.text = text;
        this.timestamp = timestamp;
    }
}
```

How do we guarantee that these Tweet objects are immutable – that, once a tweet is created, its author, message, and date can never be changed?

The first threat to immutability comes from the fact that clients can — in fact must — directly access its fields. So nothing's stopping us from writing code like this:

```
Tweet t = new Tweet("justinbieber",
                    "Thanks to all those beliebers out there inspiring me every
 day",
                    new Date());
t.author = "rbmllr";
```

This is a trivial example of **representation exposure**, meaning that code outside the class can modify the representation directly. Rep exposure like this threatens not only invariants, but also representation independence. We can't change the implementation of Tweet without affecting all the clients who are directly accessing those fields.

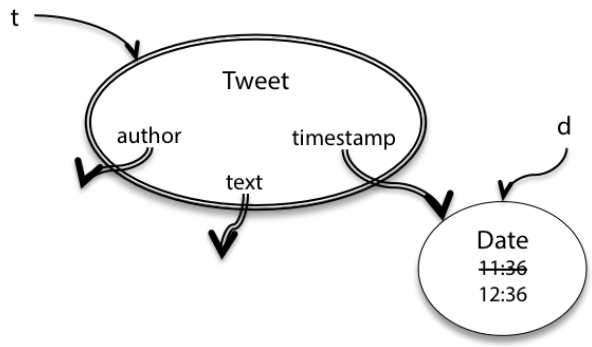
Fortunately, Java gives us language mechanisms to deal with this kind of rep exposure:

```
public class Tweet {

    private final String author;
    private final String text;
    private final Date timestamp;

    public Tweet(String author, String text, Date timestamp) {
        this.author = author;
        this.text = text;
        this.timestamp = timestamp;
    }

    /** @return Twitter user who wrote the tweet */
    public String getAuthor() {
        return author;
    }

    /** @return text of the tweet */
    public String getText() {
        return text;
    }

    /** @return date/time when the tweet was sent */
    public Date getTimestamp() {
        return timestamp;
    }

}
```

The `private` and `public` keywords indicate which fields and methods are accessible only within the class and which can be accessed from outside the class. The `final` keyword also helps by guaranteeing that the fields of this immutable type won't be reassigned after the object is constructed.

But that's not the end of the story: the rep is still exposed! Consider this perfectly reasonable client code that uses `Tweet`:

```
/** @return a tweet that retweets t, one hour later*/
public static Tweet retweetLater(Tweet t) {
    Date d = t.getTimestamp();
    d.setHours(d.getHours()+1);
    return new Tweet("rbmllr", t.getText(), d);
}
```



`retweetLater` takes a tweet and should return another tweet with the same message (called a *retweet*) but sent an hour later. The `retweetLater` method might be part of a system that automatically echoes funny things that Twitter celebrities say.

What's the problem here? The `getTimestamp` call returns a reference to the same date object referenced by tweet `t`. `t.timestamp` and `d` are aliases to the same mutable object. So when that date object is mutated by `d.setHours()`, this affects the date in `t` as well, as shown in the snapshot diagram.

`Tweet`'s immutability invariant has been broken. The problem is that `Tweet` leaked out a reference to a mutable object that its immutability depended on. We exposed the rep, in such a way that `Tweet` can no longer guarantee that its objects are immutable. Perfectly reasonable client code created a subtle bug.

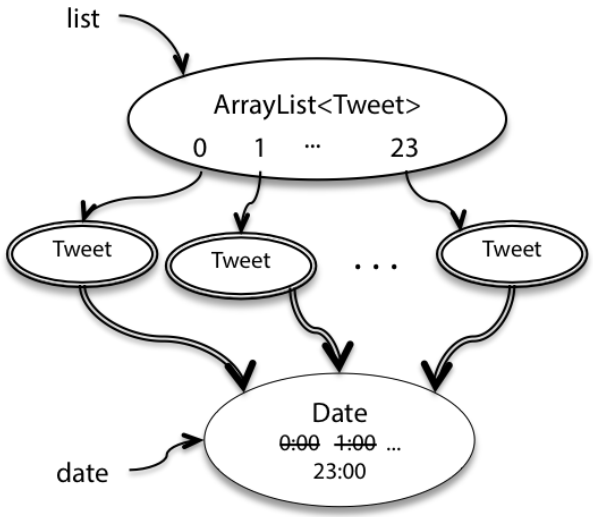
We can patch this kind of rep exposure by using defensive copying: making a copy of a mutable object to avoid leaking out references to the rep. Here's the code:

```
public Date getTimestamp() {
    return new Date(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:

```
/** @return a list of 24 inspiring tweets, one per hour today
*/
public static List<Tweet> tweetEveryHourToday () {
    List<Tweet> list = new ArrayList<Tweet>();
    Date date = new Date();
    for (int i=0; i < 24; i++) {
        date.setHours(i);
        list.add(new Tweet("rbmllr", "keep it up! you can do
 it", date));
    }
    return list;
}
```



This code intends to advance a single Date object through the 24 hours of a day, creating a tweet for every hour. But notice that the constructor of Tweet saves the reference that was passed in, so all 24 Tweet objects end up with the same time, as shown in this snapshot diagram.

Again, the immutability of Tweet has been violated. We can fix this problem too by using judicious defensive copying, this time in the constructor:

```
public Tweet(String author, String text, Date timestamp) {
    this.author = author;
    this.text = text;
    this.timestamp = new Date(timestamp.getTime());
}
```

In general, you should carefully inspect the argument types and return types of all your ADT operations. If any of the types are mutable, make sure your implementation doesn't return direct references to its representation. Doing that creates rep exposure.

You may object that this seems wasteful. Why make all these copies of dates? Why can't we just solve this problem by a carefully written specification, like this?

```
/**
 * Make a Tweet.
 * @param author    Twitter user who wrote the tweet.
 * @param text      text of the tweet
 * @param timestamp date/time when the tweet was sent. Caller must never
 *                   mutate this Date object again!
 */
public Tweet(String author, String text, Date timestamp) {
```

This approach is sometimes taken when there isn't any other reasonable alternative – for example, when the mutable object is too large to copy efficiently. But the cost in your ability to reason about the program, and your ability to avoid bugs, is enormous. In the absence of compelling arguments to the contrary, it's almost always worth it for an abstract data type to guarantee its own invariants, and preventing rep exposure is essential to that.

An even better solution is to prefer immutable types. If 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

Consider the following problematic datatype:

```
        /** 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.
 *C*         *        Requires hypotenuse^2 = legA^2 + legB^2
             *           (within the error tolerance of double arithmeti
c)
             */
            public RightTriangle(double legA, double legB, double hypote
nuse) {
/*D*/           this.sides = new double[] { legA, legB, hypotenuse };
            }

            /** Get all the sides of the triangle.
             *  @return three-element array with the triangle's side len
gths
             */
            public double[] getAllSides() {
/*E*/           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]*fact
or, 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

We now take a deeper look at the theory underlying abstract data types. This theory is not only elegant and interesting in its own right; it also has immediate practical application to the design and implementation of abstract types. If you understand the theory deeply, you'll be able to build better abstract types, and will be less likely to fall into subtle traps.

In thinking about an abstract type, it helps to consider the relationship between two spaces of values.

The space of representation values (or rep values for short) consists of the values of the actual implementation entities. In simple cases, an abstract type will be implemented as a single object, but more commonly a small network of objects is needed, so this value is actually often something rather complicated. For now, though, it will suffice to view it simply as a mathematical value.

The space of abstract values consists of the values that the type is designed to support. These are a figment of our imaginations. They're platonic entities that don't exist as described, but they are the way we want to view the elements of the abstract type, as clients of the type. For example, an abstract type for unbounded integers might have the mathematical integers as its abstract value space; the fact that it might be implemented as an array of primitive (bounded) integers, say, is not relevant to the user of the type.

Now of course the implementor of the abstract type must be interested in the representation values, since it is the implementor's job to achieve the illusion of the abstract value space using the rep value space.

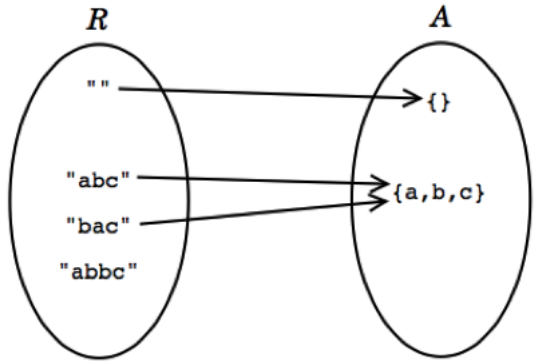
Suppose, for example, that we choose to use a string to represent a set of characters:

```
public class CharSet {
    private String s;
    ...
}
```

Then the rep space R contains Strings, and the abstract space A is mathematical sets of characters. We can show the two value spaces graphically, with an arc from a rep value to the abstract value it represents. There are several things to note about this picture:

- **Every abstract value is mapped to by some rep value**. The purpose of implementing the abstract type is to support operations on abstract values. Presumably, then, we will need to be able to create and manipulate all possible abstract values, and they must therefore be representable.

- **Some abstract values are mapped to by more than one rep value**. This happens because the representation isn't a tight encoding. There's more than one way to represent an unordered set of characters as a string.
- **Not all rep values are mapped**. In this case, the string "abbc" is not mapped. In this case, we have decided that the string should not contain duplicates. This will allow us to terminate the remove method when we hit the first instance of a particular character, since we know there can be at most one.



In practice, we can only illustrate a few elements of the two spaces and their relationships; the graph as a whole is infinite. So we describe it by giving two things:

1. An *abstraction function* that maps rep values to the abstract values they represent:

> AF : R → A

The arcs in the diagram show the abstraction function. In the terminology of functions, the properties we discussed above can be expressed by saying that the function is surjective (also called *onto*), not necessarily bijective (also called *one-to-one*), and often partial.

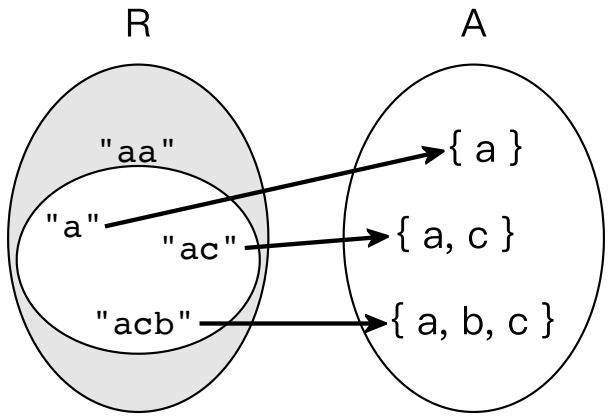
2. A *rep invariant* that maps rep values to booleans:

> 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:

```
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.

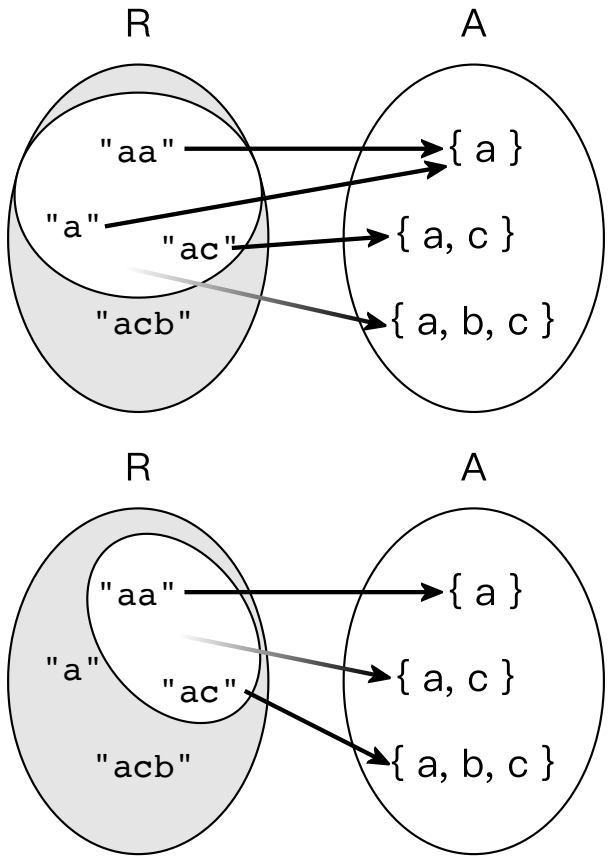
The abstract value space alone doesn't determine AF or RI: there can be several representations for the same abstract type. A set of characters could equally be represented as a string, as above, or as a bit vector, with one bit for each possible character. Clearly we need two different abstraction functions 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
    ...
}
```

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?

- [ ] abstract value space
- [ ] abstraction function
- [ ] creators
- [ ] observers
- [ ] rep
- [x] 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?

- [x] 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:

```
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`?

- [x] `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:

```
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 adjac
ent 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?

```
public void add(char c) {
    s = s + c;
}
```

✖
- [ ] SortedRep
- [ ] SortedRangeRep
- [x] 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
- [x] NoRepeatsRep ☑
- [ ] AnyRep

This implementation of `remove()` finds the first occurrence of `c` in the string and removes it.

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 `"xyyx"` 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

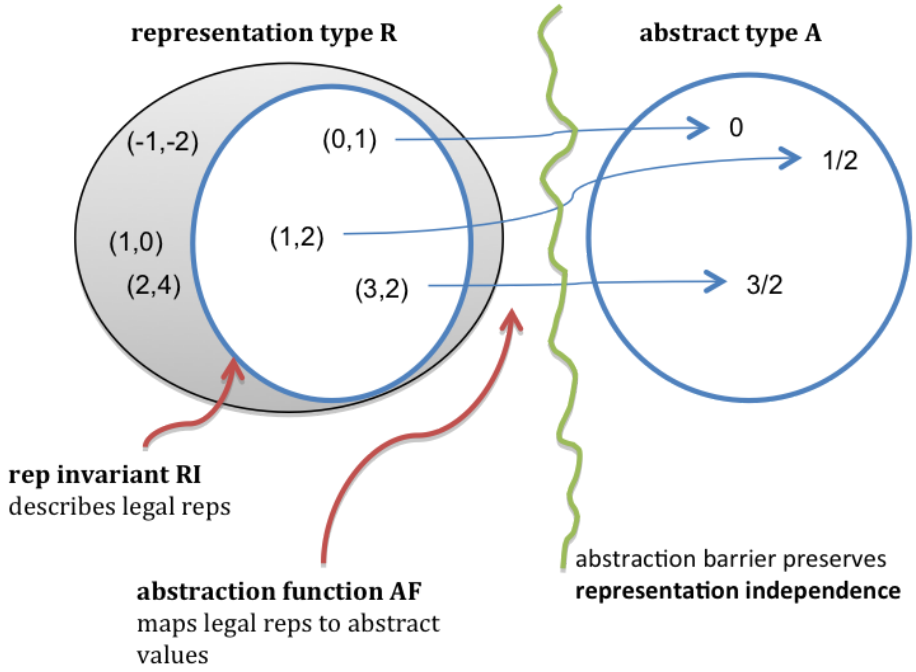
Here's an example of an abstract data type for rational numbers. Look closely at its rep invariant and abstraction function.

```
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();
```

```
    }
}
```

Here is a picture of the abstraction function and rep invariant for this code. The RI requires that numerator/denominator pairs be in reduced form (i.e., lowest terms), so pairs like (2,4) and (18,12) above should be drawn as outside the RI.

It would be completely reasonable to design another implementation of this same ADT with a more permissive RI. With such a change, some operations might become more expensive to perform, and others cheaper.



Checking the Rep Invariant

The rep invariant isn't just a neat mathematical idea. If your implementation asserts the rep invariant at run time, then you can catch bugs early. Here's a method for RatNum that tests its rep invariant:

```
// Check that the rep invariant is true
// *** Warning: this does nothing unless you turn on assertion checking
// by passing -enableassertions to Java
private void checkRep() {
    assert denom > 0;
    assert gcd(numer, denom) == 1;
}
```

You should certainly call checkRep() to assert the rep invariant at the end of every operation that creates or mutates the rep – in other words, creators, producers, and mutators. Look back at the RatNum code above, and you'll see that it calls checkRep() at the end of both constructors.

Observer methods don't normally need to call checkRep(), but it's good defensive practice to do so anyway. Why? Calling checkRep() in every method, including observers, means you'll be more likely to catch rep invariant violations caused by rep exposure.

Why is checkRep private? Who should be responsible for checking and enforcing a rep invariant – clients, or the implementation itself?

No Null Values in the Rep

Recall from the 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;
    ...
}
```

then you don't need `assert s!= null`, because the call to `s.length()` will fail just as effectively on a null reference. But if `s` is not otherwise checked by your rep invariant, then assert `s != null` explicitly.

READING EXERCISES

Checking the rep invariant

Which of the following are true?

- [ ] checkRep() is the abstraction function
- [ ] checkRep() asserts the rep invariant
- [x] 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:

```
// 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, und
erscores)
    //   text.length <= 140
    // Abstraction Function:
    //   represents a tweet posted by author, with content text, at time timesta
mp
    // 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`.

```
// 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:

```
// 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` .”

☑

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. `FollowsGraph` 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

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:

```
/** 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, side
s[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]);
    }

}
```

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.

✖ ☐ 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<Ch
aracter> set2);
```

This is easier to understand, because the name of the ADT conveys all the programmer needs to know. It's also safer from bugs, because Java static checking comes into play, and the required condition (sorted with no repeats) can be enforced in exactly one place, the `SortedSet` type.

Many of the places where we used preconditions on the problem sets would have benefited from a custom ADT instead.

READING EXERCISES

Encapsulating precondtions 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)
{ ... }
```

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

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 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, √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 → 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 ⊆ T x T that is:

- reflexive: E(t,t) ∀ t ∈ T
- symmetric: E(t,u) ⇒ E(u,t)
- transitive: E(t,u) ∧ E(u,v) ⇒ 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 ∈, these expressions are indistinguishable:

- |{1,2}| = 2 and |{2,1}| = 2
- 1 ∈ {1,2} is true, and 1 ∈ {2,1} is true
- 2 ∈ {1,2} is true, and 2 ∈ {2,1} is true
- 3 ∈ {1,2} is false, and 3 ∈ {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` ☑

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`
- [x] `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:

	referential equality	*object equality*
Java	`==`	`equals()`
Objective C	`==`	`isEqual:`
C#	`==`	`Equals()`
Python	`is`	`==`
Javascript	`==`	n/a

Note that `==` unfortunately flips its meaning between Java and Python. Don't let that confuse you: `==` in Java just tests reference identity, it doesn't compare object contents.

As programmers in any of these languages, we can't change the meaning of the referential equality operator. In Java, `==` always means referential equality. But when we define a new data type, it's our responsibility to decide what object equality means for values of the data type, and implement the `equals()` operation appropriately.

Equality of Immutable Types

The `equals()` method is defined by `Object`, and its default implementation looks like this:

```
public class Object {
    ...
    public boolean equals(Object that) {
        return this == that;
    }
}
```

In other words, the default meaning of `equals()` is the same as referential equality. For immutable data types, this is almost always wrong. So you have to **override** the `equals()` method, replacing it with your own implementation.

Here's our first try for `Duration`:

```
public class Duration {
    ...
    // Problematic definition of equals()
    public boolean equals(Duration that) {
        return this.getLength() == that.getLength();
    }
}
```

There's a subtle problem here. Why doesn't this work? Let's try this code:

```
Duration d1 = new Duration (1, 2);
Duration d2 = new Duration (1, 2);
Object o2 = d2;
d1.equals(d2) → true
d1.equals(o2) → false
```

You can see this code in action (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%2
frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curInstr=33). You'll see that even though `d2` and `o2` end up referring to the very same object in memory, you still get different results for them from `equals()`.

What's going on? It turns out that `Duration` has **overloaded** the `equals()` method, because the method signature was not identical to `Object`'s. We actually have two `equals()` methods in `Duration`: an implicit `equals(Object)` inherited from `Object`, and the new `equals(Duration)`.

```
public class Duration extends Object {
    // explicit method that we declared:
    public boolean equals (Duration that) {
        return this.getLength() == that.getLength();
    }
    // implicit method inherited from Object:
    public boolean equals (Object that) {
        return this == that;
    }
}
```

We've seen overloading since the very beginning of the course in static checking (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.

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
(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,+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%2
frontend.js&cumulative=false&heapPrimitives=false&textReferences=false&py=java&rawInputLstJSON=%5B%5D&curInstr=49) in the Online Python Tutor.

The Object Contract

The specification of the `Object` class is so important that it is often referred to as *the `Object` Contract*. The contract can be found in the method specifications for the `Object` class. Here we will focus on the contract for `equals`. When you override the `equals` method, you must adhere to its general contract. It states that:

- `equals` must define an equivalence relation – that is, a relation that is reflexive, symmetric, and transitive;
- `equals` must be consistent: repeated calls to the method must yield the same result provided no information used in `equals` comparisons on the object is modified;
- for a non-null reference `x`, `x.equals(null)` should return false;
- `hashCode` must produce the same result for two objects that are deemed equal by the `equals` method.

Breaking the Equivalence Relation

Let's start with the equivalence relation. We have to make sure that the definition of equality implemented by `equals()` is actually an equivalence relation as defined earlier: reflexive, symmetric, and transitive. If it isn't, then operations that depend on equality (like sets, searching) will behave erratically and unpredictably. You don't want to program with a data type in which sometimes `a` equals `b`, but `b` doesn't equal `a`. Subtle and painful bugs will result.

Here's an example of how an innocent attempt to make equality more flexible can go wrong. Suppose we wanted to allow for a tolerance in comparing `Duration` objects, because different computers may have slightly unsynchronized clocks:

```
private static final int CLOCK_SKEW = 5; // seconds

@Override
public boolean equals (Object thatObject) {
    if (!(thatObject instanceof Duration)) return false;
    Duration thatDuration = (Duration) thatObject;
    return Math.abs(this.getLength() - thatDuration.getLength()) <= CLOCK_SKEW;
}
```

Which property of the equivalence relation is violated?

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)` ☑

- [x] `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.)

- [x] recursivity
- [ ] reflexivity
- [x] 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.

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 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 mrked `TODO` to make `hashCode()` consistent with `equals()`?

- [x] `return 42;`
- [x] `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.

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%(List%3CString%3E+l+%3A+set%29+%7B+%0A++++++System.out.println(%22set.contains(l%29%3D%22+%2B+set.contains(l%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&curInstr=13) on Online Python Tutor.

What's going on? `List<String>` is a mutable object. In the standard Java implementation of collection classes like `List`, mutations affect the result of `equals()` and `hashCode()`. When the list is first put into the `HashSet`, it is stored in the hash bucket corresponding to its `hashCode()` result at that time. When the list is subsequently mutated, its `hashCode()` changes, but `HashSet` doesn't realize it should be moved to a different bucket. So it can never be found again.

When `equals()` and `hashCode()` can be affected by mutation, we can break the rep invariant of a hash table that uses that object as a key.

Here's a telling quote from the specification of `java.util.Set`:

> Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set.

The Java library is unfortunately inconsistent about its interpretation of `equals()` for mutable classes. Collections use observational equality, but other mutable classes (like `StringBuilder`) use behavioral equality.

The lesson we should draw from this example is that **`equals()` should implement behavioral equality**. In general, that means that two references should be `equals()` if and only if they are aliases for the same object. So mutable objects should just inherit `equals()` and `hashCode()` from `Object`. For clients that need a notion of observational equality (whether two mutable objects "look" the same in the current state), it's better to define a new method, e.g., `similar()`.

The Final Rule for equals and hashCode()

For immutable types:

- `equals()` should compare abstract values. This is the same as saying `equals()` should provide behavioral equality.
- `hashCode()` should map the abstract value to an integer.

So immutable types must override both `equals()` and `hashCode()`.

For mutable types:

- `equals()` should compare references, just like `==`. Again, this is the same as saying `equals()` should provide behavioral equality.
- `hashCode` should map the reference into an integer.

So mutable types should not override `equals()` and `hashCode()` at all, and should simply use the default implementations provided by `Object`. Java doesn't follow this rule for its collections, unfortunately, leading to the pitfalls that we saw above.

READING EXERCISES

You are not logged in.

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 occurence of e, and return this bag */
public Bag<E> add(E e)

/** modify this bag by removing an occurence 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<>().add("a").add("b");
Bag<String> b2 = new Bag<>().add("a").add("b");
Bag<String> b3 = b1.remove("b");
Bag<String> b4 = new Bag<>().add("b").add("a"); // swap!
```

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` ☑
- [x] `b2.count("b") == 1` ☑
- [ ] `b3.count("a") == 1` ☑
- [ ] `b3.count("b") == 1`
- [ ] `b4.count("a") == 1` ☑

- [x] `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 occurence 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)`
- [x] `b1.equals(b4)`
- [ ] `b2.equals(b3)`
- [ ] `b2.equals(b4)`
- [x] `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)`
- [x] `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:

```
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&curInstr=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.

Reading 19: Concurrency

Software in 6.005

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

Objectives

- The message passing and shared memory models of concurrency
- Concurrent processes and threads, and time slicing
- The danger of race conditions

Concurrency

Concurrency means multiple computations are happening at the same time. Concurrency is everywhere in modern programming, whether we like it or not:

- Multiple computers in a network
- Multiple applications running on one computer
- Multiple processors in a computer (today, often multiple processor cores on a single chip)

In fact, concurrency is essential in modern programming:

- Web sites must handle multiple simultaneous users.
- Mobile apps need to do some of their processing on servers ("in the cloud").
- Graphical user interfaces almost always require background work that does not interrupt the user. For example, Eclipse compiles your Java code while you're still editing it.

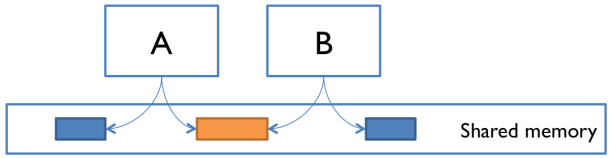
Being able to program with concurrency will still be important in the future. Processor clock speeds are no longer increasing. Instead, we're getting more cores with each new generation of chips. So in the future, in order to get a computation to run faster, we'll have to split up a computation into concurrent

pieces.

Two Models for Concurrent Programming

There are two common models for concurrent programming: *shared memory* and *message passing*.

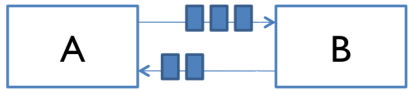
Shared memory. In the shared memory model of concurrency, concurrent modules interact by reading and writing shared objects in memory.



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.

The process abstraction is a *virtual computer*. It makes the program feel like it has the entire machine to itself – like a fresh computer has been created, with fresh memory, just to run that program.

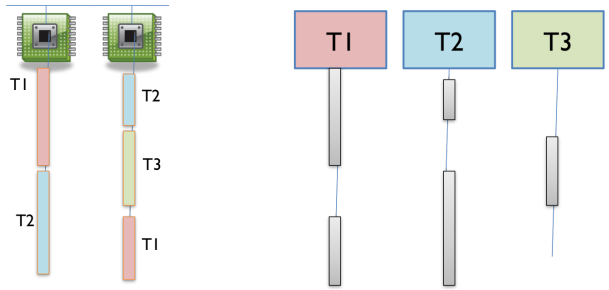
Just like computers connected across a network, processes normally share no memory between them. A process can't access another process's memory or objects at all. Sharing memory between processes is *possible* on most operating systems, but it needs special effort. By contrast, a new process is automatically ready for message passing, because it is created with standard input & output streams, which are the `System.out` and `System.in` streams you've used in Java.

Thread. A thread is a locus of control inside a running program. Think of it as a place in the program that is being run, plus the stack of method calls that led to that place (so the thread can go back up the stack when it reaches `return` statements).

Just as a process represents a virtual computer, the thread abstraction represents a *virtual processor*. Making a new thread simulates making a fresh processor inside the virtual computer represented by the process. This new virtual processor runs the same program and shares the same memory as other threads in the process.

Threads are automatically ready for shared memory, because threads share all the memory in the process. It takes special effort to get "thread-local" memory that's private to a single thread. It's also necessary to set up message-passing explicitly, by creating and using queue data structures. We'll talk about how to do that in a future reading.

How can I have many concurrent threads with only one or two processors in my computer? When there are more threads than processors, concurrency is simulated by **time slicing**, which means that the processor switches between threads. The figure on the right shows how three threads T1, T2, and T3 might be time-sliced on a machine that has only two actual processors. In the figure, time proceeds downward, so at first one processor is running thread T1 and the other is running thread T2, and then the second processor switches to run thread T3. Thread T2 simply pauses, until its next time slice on the same processor or another processor.



On most systems, time slicing happens unpredictably and nondeterministically, meaning that a thread may be paused or resumed at any time.

In the Java Tutorials, read:

- **Processes & Threads (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 (subclassing `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.

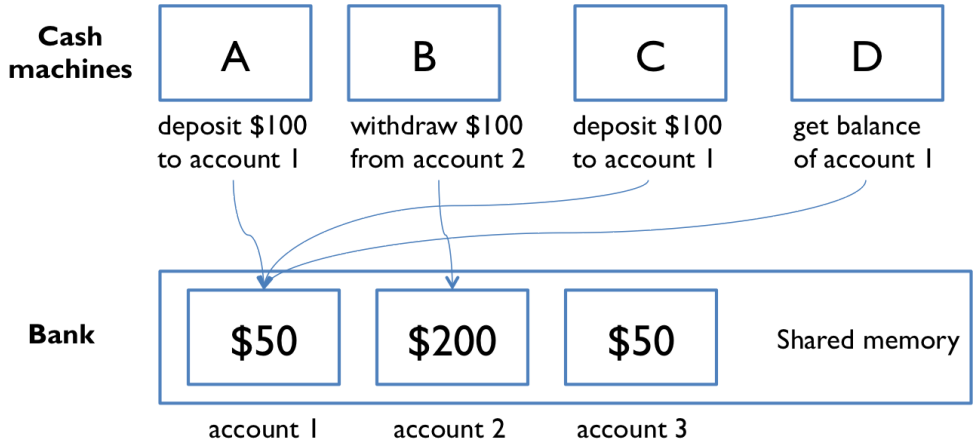
When a program starts, it automatically gets one process to run in, and it gets one thread for its execution.

CHECK EXPLAIN

Shared Memory Example

Let's look at an example of a shared memory system. The point of this example is to show that concurrent programming is hard, because it can have subtle bugs.

Imagine that a bank has cash machines that use a shared memory model, so all the cash machines can read and write the same account objects in memory.



To illustrate what can go wrong, let's simplify the bank down to a single account, with a dollar balance stored in the `balance` variable, and two operations `deposit` and `withdraw` that simply add or remove a dollar:

```
// suppose all the cash machines share a single bank account
private static int balance = 0;

private static void deposit() {
    balance = balance + 1;
}
private static void withdraw() {
    balance = balance - 1;
}
```

Customers use the cash machines to do transactions like this:

```
deposit(); // put a dollar in
withdraw(); // take it back out
```

In this simple example, every transaction is just a one dollar deposit followed by a one-dollar withdrawal, so it should leave the balance in the account unchanged. Throughout the day, each cash machine in our network is processing a sequence of deposit/withdraw transactions.

```
// each ATM does a bunch of transactions that
// modify balance, but leave it unchanged afterward
private static void cashMachine() {
    for (int i = 0; i < TRANSACTIONS_PER_MACHINE; ++i) {
        deposit(); // put a dollar in
        withdraw(); // take it back out
    }
}
```

So at the end of the day, regardless of how many cash machines were running, or how many transactions we processed, we should expect the account balance to still be 0.

But if we run this code, we discover frequently that the balance at the end of the day is *not* 0. If more than one `cashMachine()` call is running at the same time – say, on separate processors in the same computer – then `balance` may not be zero at the end of the day. Why not?

Interleaving

Here's one thing that can happen. Suppose two cash machines, A and B, are both working on a deposit at the same time. Here's how the `deposit()` step typically breaks down into low-level processor instructions:

get balance (balance=0)
add 1
write back the result (balance=1)

When A and B are running concurrently, these low-level instructions interleave with each other (some might even be simultaneous in some sense, but let's just worry about interleaving for now):

A	B
A get balance (balance=0)	
A add 1	
A write back the result (balance=1)	
	B get balance (balance=1)
	B add 1
	B write back the result (balance=2)

This interleaving is fine – we end up with balance 2, so both A and B successfully put in a dollar. But what if the interleaving looked like this:

A	B
A get balance (balance=0)	
	B get balance (balance=0)
A add 1	
	B add 1
A write back the result (balance=1)	
	B write back the result (balance=1)

The balance is now 1 – A's dollar was lost! A and B both read the balance at the same time, computed separate final balances, and then raced to store back the new balance – which failed to take the other's deposit into account.

Race Condition

This is an example of a **race condition**. A race condition means that the correctness of the program (the satisfaction of postconditions and invariants) depends on the relative timing of events in concurrent computations A and B. When this happens, we say "A is in a race with B."

Some interleavings of events may be OK, in the sense that they are consistent with what a single, nonconcurrent process would produce, but other interleavings produce wrong answers – violating postconditions or invariants.

Tweaking the Code Won't Help

All these versions of the bank-account code exhibit the same race condition:

```
// version 1
private static void deposit() {
    balance = balance + 1;
}
private static void withdraw() {
    balance = balance - 1;
}
```

```
// version 2
private static void deposit() {
    balance += 1;
}
private static void withdraw() {
    balance -= 1;
}
```

```
// version 3
private static void deposit() {
    ++balance;
}
private static void withdraw() {
    --balance;
}
```

You can't tell just from looking at Java code how the processor is going to execute it. You can't tell what the indivisible operations – the atomic operations – will be. It isn't atomic just because it's one line of Java. It doesn't touch balance only once just because the balance identifier occurs only once in the

line. The Java compiler, and in fact the processor itself, makes no commitments about what low-level operations it will generate from your code. In fact, a typical modern Java compiler produces exactly the same code for all three of these versions!

The key lesson is that you can't tell by looking at an expression whether it will be safe from race conditions.

> Read: **Thread Interference (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:

```
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, `tmpr` and `tmpa`, to manipulate the fields `ready` and `answer`:

```
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

If `methodA` runs first, then it sets `x` to 1×2×3 = 6, and then `methodB` runs and sets `x` to 5×6 = 30. Since multiplication is commutative, we get the same result if `methodB` runs before `methodA`.

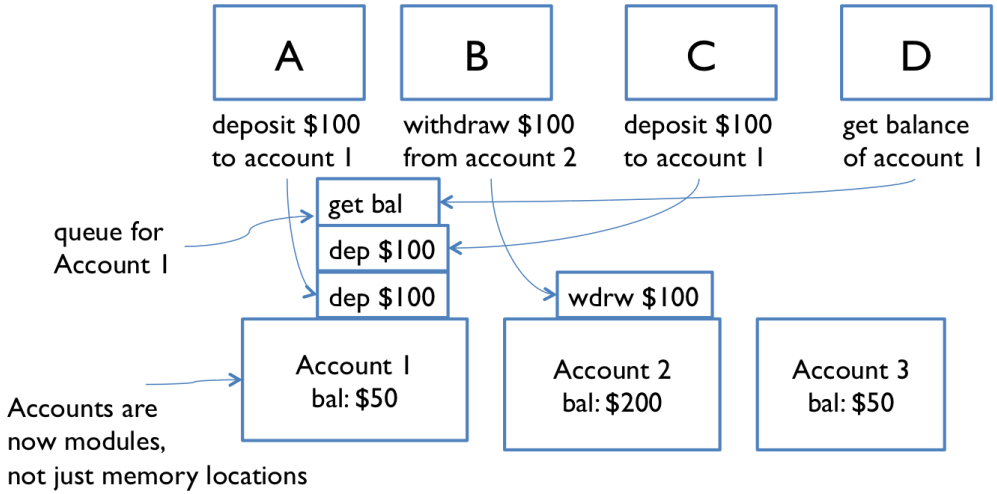
CHECK EXPLAIN

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

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.

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'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

You are not logged in.

Collaboratively authored with contributions from: Jamie Brandon, Chanseok Oh, Mardavij Roozbehani, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama.

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.

Reading 20: Thread Safety

Software in 6.005

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

Objectives

Recall race conditions: multiple threads sharing the same mutable variable without coordinating what they're doing. This is unsafe, because the correctness of the program may depend on accidents of timing of their low-level operations.

There are basically four ways to make variable access safe in shared-memory concurrency:

- **Confinement.** Don't share the variable between threads. This idea is called confinement, and we'll explore it today.
- **Immutability.** Make the shared data immutable. We've talked a lot about immutability already, but there are some additional constraints for concurrent programming that we'll talk about in this reading.
- **Threadsafe data type.** Encapsulate the shared data in an existing threadsafe data type that does the coordination for you. We'll talk about that today.
- **Synchronization.** Use synchronization to keep the threads from accessing the variable at the same time. Synchronization is what you need to build your own threadsafe data type.

We'll talk about the first three ways in this reading, along with how to make an argument that your code is threadsafe using those three ideas. We'll talk about the fourth approach, synchronization, in a later reading.

The material in this reading is inspired by an excellent book: Brian Goetz et al., *Java Concurrency in Practice (http://jcip.net/)*, Addison-Wesley, 2006.

What Threadsafe Means

A data type or static method is *threadsafe* if it behaves correctly when used from multiple threads, regardless of how those threads are executed, and without demanding additional coordination from the calling code.

- "behaves correctly" means satisfying its specification and preserving its rep invariant;
- "regardless of how threads are executed" means threads might be on multiple processors or timesliced on the same processor;
- "without additional coordination" means that the data type can't put preconditions on its caller related to timing, like "you can't call `get()` while `set()` is in progress."

Remember `Iterator` (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:

```
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);
    }
}
```

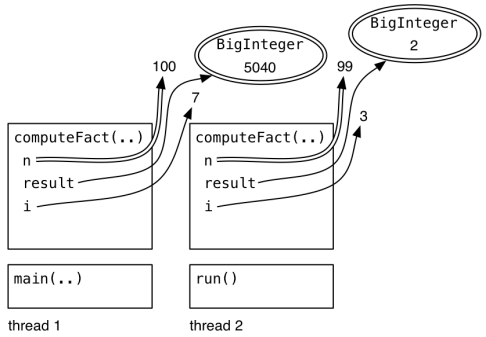
Starting a thread with an **anonymous (https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html)** `Runnable` is a common idiom. Read:

- 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:

1. When we start the program, we start with one thread running `main`.
2. `main` creates a second thread using the anonymous `Runnable` idiom, and starts that thread.
3. At this point, we have two concurrent threads of execution. Their interleaving is unknown! But one *possibility* for the next thing that happens is that thread 1 enters `computeFact`.
4. Then, the next thing that *might* happen is that thread 2 also enters `computeFact`.

   At this point, we see how **confinement** helps with thread safety: each execution of `computeFact` has its own `n`, `i`, and `result` variables. None of the objects they point to are mutable; if they were mutable, we would need to check that the objects are not aliased from other threads.
5. The `computeFact` computations proceed independently, updating their respective variables



Avoid Global Variables

Unlike local variables, static variables are not automatically thread confined.

If you have static variables in your program, then you have to make an argument that only one thread will ever use them, and you have to document that fact clearly. Better, you should eliminate the static variables entirely.

Here's an example:

```
// This class has a race condition in it.
public class PinballSimulator {

    private static PinballSimulator simulator = null;
    // invariant: there should never be more than one PinballSimulator
    //            object created

    private PinballSimulator() {
        System.out.println("created a PinballSimulator object");
    }

    // factory method that returns the sole PinballSimulator object,
    // creating it if it doesn't exist
    public static PinballSimulator getInstance() {
        if (simulator == null) {
            simulator = new PinballSimulator();
        }
        return simulator;
    }
}
```

This class has a race in the `getInstance()` method – two threads could call it at the same time and end up creating two copies of the `PinballSimulator` object, which we don't want.

To fix this race using the thread confinement approach, you would specify that only a certain thread (maybe the "pinball simulation thread") is allowed to call `PinballSimulator.getInstance()`. The risk here is that Java won't help you guarantee this.

In general, static variables are very risky for concurrency. They might be hiding behind an innocuous function that seems to have no side-effects or mutations. Consider this example:

```
// is this method threadsafe?
/**
 * @param x integer to test for primeness; requires x > 1
 * @return true if 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:

```
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?

- [x] 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:

```
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

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/recursive/#tuning_the_rep) the first time the length was requested by a client. Caching is a typical kind of beneficent mutation.

For concurrency, though, this kind of hidden mutation is not safe. An immutable data type that uses beneficent mutation will have to make itself threadsafe using locks (the same technique required of mutable data types), which we'll talk about in a future reading.

Stronger definition of immutability

So in order to be confident that an immutable data type is threadsafe without locks, we need a stronger definition of immutability:

- no mutator methods
- all fields are private and final
- no representation exposure (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/recursive/#tuning_the_rep)

If you follow these rules, then you can be confident that your immutable type will also be threadsafe.

In the Java Tutorials, read:

- **A Strategy for Defining Immutable Objects (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 ☑
- [x] client calls to creators
- [ ] producer implementations ☑

- [ ] client calls to producers
- [x] 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:

> [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.

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 lis
t wrapper
```

The solution to this iteration problem will be to acquire the collection's lock when you need to iterate over it, which we'll talk about in a future reading.

Finally, **atomic operations aren't enough to prevent races:** the way that you use the synchronized collection can still have a race condition. Consider this code, which checks whether a list has at least one element and then gets that element:

```
if ( ! lst.isEmpty()) { String s = lst.get(0); ... }
```

Even if you make `lst` into a synchronized list, this code still may have a race condition, because another thread may remove the element between the `isEmpty()` call and the `get()` call.

Even the `isPrime()` method still has potential races:

```
if (cache.containsKey(x)) return cache.get(x);
boolean answer = BigInteger.valueOf(x).isProbablePrime(100);
cache.put(x, answer);
```

The synchronized map ensures that `containsKey()`, `get()`, and `put()` are now atomic, so using them from multiple threads won't damage the rep invariant of the map. But those three operations can now interleave in arbitrary ways with each other, which might break the invariant that `isPrime` needs from the cache: if the cache maps an integer x to a value f, then x is prime if and only if f is true. If the cache ever fails this invariant, then we might return the wrong result.

So we have to argue that the races between `containsKey()`, `get()`, and `put()` don't threaten this invariant.

1. The race between `containsKey()` and `get()` is not harmful because we never remove items from the cache – once it contains a result for x, it will continue to do so.
2. There's a race between `containsKey()` and `put()`. As a result, it may end up that two threads will both test the primeness of the same x at the same time, and both will race to call `put()` with the answer. But both of them should call `put()` with the same answer, so it doesn't matter which one wins the race – the result will be the same.

The need to make these kinds of careful arguments about safety – even when you're using threadsafe data types – is the main reason that concurrency is hard.

In the Java Tutorials, read:

- **Wrapper Collections (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

Threadsafe data types

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` ☑
- [x] `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` ☑

✖
- [x] `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 that they use, and argue which of the four techniques you are using to protect against races for each data object or variable: confinement, immutability, threadsafe data types, or synchronization. When you use the last two, you also need to argue that all accesses to the data are appropriately atomic – that is, that the invariants you depend on are not threatened by interleaving. We gave one of those arguments for `isPrime` above.

Thread Safety Arguments for Data Types

Let's see some examples of how to make thread safety arguments for a data type. Remember our four approaches to thread safety: confinement, immutability, threadsafe data types, and synchronization. Since we haven't talked about synchronization in this reading, we'll just focus on the first three approaches.

Confinement is not usually an option when we're making an argument just about a data type, because you have to know what threads exist in the system and what objects they've been given access to. If the data type creates its own set of threads, then you can talk about confinement with respect to those threads. Otherwise, the threads are coming in from the outside, carrying client calls, and the data type may have no guarantees about which threads have references to what. So confinement isn't a useful argument in that case. Usually we use confinement at a higher level, talking about the system as a whole and arguing why we don't need thread safety for some of our modules or data types, because they won't be shared across threads by design.

Immutability is often a useful argument:

```
/** MyString is an immutable data type representing a string of characters. */
public class MyString {
    private final char[] a;
    // Thread safety argument:
    //    This class is threadsafe because it's immutable:
    //    - a is final
    //    - a points to a mutable char array, but that array is encapsulated
    //      in this object, not shared with any other object or exposed to a
    //      client
```

Here's another rep for MyString that requires a little more care in the argument:

```
/** MyString is an immutable data type representing a string of characters. */
public class MyString {
    private final char[] a;
    private final int start;
    private final int len;
    // Rep invariant:
    //    0 <= start <= a.length
    //    0 <= len <= a.length-start
    // Abstraction function:
    //    represents the string of characters a[start],...,a[start+length-1]
    // Thread safety argument:
    //    This class is threadsafe because it's immutable:
    //    - a, start, and len are final
    //    - a points to a mutable char array, which may be shared with other
    //      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:

```
/** 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

`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:

```
/** 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 "";
        }
```

```
    }
}
```

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`
- [x] `clear`
- [ ] `first`

Both `insert` and `first` may fail if `clear` interleaves at the wrong moment – they will throw `IndexOutOfBounds` 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")`

run `clear` and `insert` concurrently as shown on the right.

A	B
— in `clear: text = ""`	
	— in `insert: text = "" + "a" + "z"`
— `clear` returns	
	— `insert` returns
`assert sb.toString() .equals("")`	

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	B
call `sb.clear()`	
— in `clear: text = ""`	
— `clear` returns	
	call `sb.insert(0, "a")`
	— in `insert: text = "" + "a" + "z"`
	— `insert` returns
`assert sb.toString() .equals("")`	

What we demand from a threadsafe data type is that when clients call its atomic operations concurrently, the results are consistent with *some* sequential ordering of the calls. In this case, clearing and inserting, that means either `clear`-followed-by-`insert`, or `insert`-followed-by-`clear`. This

property is called **serializability** (https://en.wikipedia.org/wiki/Serializability): for any set of operations executed concurrently, the result (the values and state observable by clients) must be a result given by *some* sequential ordering of those operations.

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.

◯ 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.