L8: Mutability & Immutability

Today

- mutable objects
- aliasing
- immutability

Required reading (from the Java Tutorial and codeguru)

- Classes and Objects (~20 pages)
  http://docs.oracle.com/javase/tutorial/java/javaOO/index.html
- the static keyword (1 page)
  http://www.codeguru.com/java/tij/tij0037.shtml#Heading79
- the final keyword (1 page)
  http://www.codeguru.com/java/tij/tij0071.shtml

Snapshot Diagrams

It will be useful for us to draw pictures of what’s happening at runtime, in order to understand subtle issues. **Snapshot diagrams** represent the internal state of a program at runtime – its stack (methods in progress and their local variables) and its heap (objects that currently exist).

Here’s why we use snapshot diagrams in 6.005:

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

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

**Primitive values**

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

```
3  5.0  'c'  null
```
Object values

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

Mutating Values vs. Reassigning Variables

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

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

Immutability (immunity from change) is a major design principle in this course. Immutable types are types whose values can never change once they have been created.

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

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

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

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

In a snapshot diagram, an immutable reference (`final`) is denoted by a crossbar just behind the arrowhead. Here’s an object whose `id` never changes (it can’t be reassigned to a different number), but whose `age` can change.

Immutable Objects vs. Mutable Objects

String is immutable: once created, a String object always has the same value. To add something to the end of a String, you have to create a new String object:
String s = “a”;
s = s.concat(“b”);    /// s = s + “b” is syntactic sugar for this call

Immutable objects (intended by their designer to always represent the same value) are denoted by a double border. For example, here’s an Integer object, the result of new Integer(7):

By design, this Integer object can never change value during its lifetime. There is no method on it that will change it to a different integer value.

By contrast, StringBuilder (another built-in Java class) is a mutable object that represents a string of characters. It has operations that change the value of the object, rather than just returning new values:

StringBuilder sb = new StringBuilder(“a”);
    sb.append(“b”);

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

So what? In both cases, you end up with s and sb referring to the string of characters abcdef. The difference between mutability and immutability doesn’t matter much when there’s only one reference to the object. But there are big differences in how they behave when there are other references to the object, like t and tb introduced below:

String t = s;
t = t + “c”;

StringBuilder tb = sb;
tb.append(“c”);
Why do we need the mutable StringBuilder in programming? A common use for it is to concatenate a large number of strings together, like this:

```java
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:

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

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

But the convenience of mutable data comes with big risks. Mutability makes it harder to understand what your program is doing, and much harder to enforce contracts. We’ll see an example of that later in the lecture.

**Arrays and Lists**

Like other object values, arrays and lists are labeled with their type. In lieu of field names, we label the outgoing edges with indexes \((0, 1, ...)\). When the sequence of elements is obvious, we may omit the index labels.
For example, here’s how we would draw the data structures produced by our two variants of the hailstone sequence, when the starting \( n \) is 7:

A couple of things are worth noting:

- the array is fixed size, always 100 elements. Java happens to initialize the extra entries at the end of the array to 0 (that’s a requirement of the Java language definition), but arrays in many languages do not have that guarantee, and you may start out with completely arbitrary garbage values at the end of your array. So it’s good not to depend on automatic initialization when you program.
- the ArrayList only has as many elements as the hailstone sequence code actually added.
- both kinds of sequences are indexed starting from 0.
- the array points to primitive int values, but the ArrayList actually points to Integer objects that in turn refer to primitive int values. We will sometimes omit this distinction in future pictures of ArrayList\(<\text{Integer}\>\), and just draw arrows straight to ints. But for now here’s the gory detail.
- the Integer objects are immutable (shown by the double border) and the references from them are also immutable (shown by a stroke across the arrow). Once an Integer object is created for a particular integer value, it represents that value for its entire lifetime. More on immutability in the next section.

Both the array object and the ArrayList\(<\text{Integer}\>\) object are mutable, as indicated by the single-line border. How do we know? Because references from the numbered cells (0...99 in the array, 0..16 in the ArrayList) can be reassigned to point to other values, which changes the overall value of the sequence of integers that the array or list represents.
Specification for a Mutating Method

Let's go back to specifications for a bit, and talk about how to describe mutations in the postcondition. Here's a specification that describes a method that mutates an object:

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

We've taken this, slightly simplified, from the Java List interface. First, look at the postcondition. It gives two constraints: the first telling us how list1 is modified, and the second telling us how the return value is determined.

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

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

Here is another example of a mutating method:

```java
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:

```java
static List<String> toLowerCase(List<String> lst)
    requires: nothing
    effects: returns a new list t where t[i] = lst[i].toLowerCase(). Doesn't modify lst.
```

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:

```java
List<String> l = ...;
for (String s : l) {
    System.out.println(s);
}
```

is rewritten by the compiler into something like this:

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

An iterator has two methods:

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

Note that next() 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.

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

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

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

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

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

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

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

- **instance variables**, also called fields in Java. Instance variables differ from method parameters and local variables; the instance variables are stored in the object instance and persist for longer than a method call. What are the instance variables of MyIterator?

- a **constructor**, which makes a new object instance and initializes its instance variables. Where is the constructor of MyIterator?

- the **static** keyword is missing from MyIterator’s methods, which means they are instance methods that must be called on an instance of the object, e.g. `iter.next()`.

- the **this** keyword is used at one point to refer to the instance object, in particular to refer to an instance variable (`this.l`). This was done to disambiguate two different variables named `l` (an instance variable and a constructor parameter). Most of MyIterator’s code refers to instance variables without an explicit `this`, but this is just a convenient shorthand that Java supports -- e.g., `i` actually means `this.i`.

- **private** is used for the object's internal state and internal helper methods, while **public** indicates methods and constructors that are intended for clients of the class.

- **final** is used to indicate which parts of the object’s internal state can change and which can’t. `i` is allowed to change (`next()` updates it as it steps through the list), but `l` cannot (the iterator has to keep pointing at the same list for its entire life – if you want to iterate through another list, you’re expected to create another iterator object).

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

![Snapshot Diagram](image)

Note that we drew a slash across the arrow from `l`, to indicate that it’s **final**. That means that the arrow can’t change once it’s drawn. But the ArrayList object it points to is mutable – elements can be changed within it -- and declaring `l` as final has no effect on that.

Why do iterators exist? There are many kinds of collection data structures (linked lists, maps, hash tables) with different kinds of internal representations. The iterator concept allows a single uniform way to access them all, so that client code is simpler and the collection implementation can change without changing the client code 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.
Regarding a mutable object as a state machine

One useful perspective for regarding a mutable object is as a state machine. A state machine is a set of states that a system can be in (drawn as boxes) with the possible transitions between them (drawn as edges). The transitions are called events.

For a mutable object, the state is represented by the instance variables of the object – particularly the ones that can be mutated over the course of the object's lifetime. The state of a MyIterator object is represented by $i$, the index of its current position in the list.

The events are the public operations that can be performed on the object -- the methods that can be called. These represent events arriving from the outside world (from a client using the object) that cause transitions in the object's internal state. For a MyIterator, the methods are hasNext() and next(). Of these two methods, next() is a mutator, however, so it can cause transitions to a new state. hasNext() is an observer – it returns information about the current state, but never causes a transition to a new state.

Here's a very explicit state machine diagram for a MyIterator object over a list of length $n$ (for $n > 0$), showing all the possible states that the object can experience:

In practice we will find it more useful to draw more abstract state diagrams that combine multiple states together:

Here we've combined all the states where $i < n$ into a single state INPROGRESS, and combined their transitions as well. Note that we now have a nondeterministic state machine -- the next event (method call) can transition to either INPROGRESS (if we still have more elements to return) or to DONE (if we've run to the end of the list). In this case, the nondeterminism was introduced by our decision to abstract away from the particular value of $i$. The behavior of the MyIterator object is still completely deterministic; it knows which particular value of $i$ it has. We've just omitted it from this picture. Other state machines might be genuinely nondeterministic, in the sense that the transition chosen in response to an event might be random.

This state machine also correctly handles the case where $n$ (the length of the list) is zero, by having two initial-state transitions. Either INPROGRESS or DONE may be the starting state of the machine.

We can further elaborate this state machine by attaching outputs to each transition. Since each event is modeling a method call, the output specifies the result of the call – a return value if the method
returns normally, or an exception if it throws an exception. For MyIterator, for example, the return value of hasNext is determined by the state:

With outputs, we can use this compact diagram to describe different designs for the class. Here are two alternative designs for MyIterator:

What’s the difference between these and the spec we’ve been using? How would you change the comments in MyIterator’s Java code to describe these?

The risk of mutation

Let’s try using our iterator for a simple job. Suppose we have a list of tokens extracted from a web page, so the tokens include both words (“hello” and “world”) and HTML tags (“<a>”, “<br>”, etc.). We want a method stripTags that will delete the HTML tags from the list, leaving the words behind. Following good practices, we first write the spec:

```java
/**
 * Remove html tags from a list.
 * Modifies l by removing elements of the form "<*>".
 * @param l list of words and html tags.
 */
public static void stripTags(ArrayList<String> l) {...}
```

Note that stripTags has a frame condition (modifies clause) in its contract, warning the client that its list argument will be mutated.
Next, following test-first programming, we devise a testing strategy that partitions the input space, and choose test cases to cover that partition:

// Testing strategy:
// l.size: 0, 1, n
// contents: no tags, one tag, all tags
// position: tag at start, tag in middle, tag at end
// kind of element: <foo>, </foo>, word, empty string

// Test cases:
// [] => []
// ["a"] => ["a"]
// ["a", "b", "c"] => ["a", "b", "c"]
// ["a", "<b>", "c"] => ["a", "c"]
// ["<a>", "<b>", "<c>"] => []

Finally we implement it:

```java
public static void stripTags(ArrayList<String> l) {
    MyIterator iter = new MyIterator(l);
    while (iter.hasNext()) {
        String s = iter.next();
        if (isTag(s)) {
            l.remove(s);
        }
    }
}
```

```java
private static boolean isTag(String s) {
    return s.startsWith("<") && s.endsWith(">");
}
```

Note that we pulled out the test for whether a token is HTML into a separate method, isTag. This improves the readability of stripTags, and allows us to make isTag more complicated if necessary, since HTML can be very complicated, and test it independently of stripTags.

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

```
// stripTags(["<a>", "<b>", "<c>"])
// expected [], actual ["<b>"]
```

We got the wrong answer: stripTags left a tag behind in the list. Why? Trace through what happens. It will help to use a snapshot diagram showing the MyIterator object and the ArrayList object and update it while you work through the code.

Note that this isn’t just a bug in our MyIterator. The built-in iterator in ArrayList suffers from the same problem, and so does the for() loop that’s syntactic sugar for it. The problem just has a different symptom. If you used this code instead:

```java
for (String s : l) {
    if (isTag(s)) {
        l.remove(s);
    }
}
```

then you’ll get a ConcurrentModificationException. The builtin iterator detects that you’re changing the list under its feet, and cries foul. (How do you think it does that?)
How can you fix this problem? One way is to use the remove() method of Iterator, so that the iterator adjusts its index appropriately:

```java
Iterator iter = l.iterator();
while (iter.hasNext()) {
    String s = iter.next();
    if (isTag(s)) {
        iter.remove(s);
    }
}
```

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

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

**Aliasing**

This is a fundamental issue with mutable data structures. Multiple references to the same mutable object (also called aliases for the object) may mean that multiple places in your program – possibly widely separated – are relying on that object to remain consistent.

To put it in terms of specifications, contracts can’t be enforced in just one place anymore, e.g. between the client of a class and the implementer of a class. Contracts involving mutable objects now depend on the good behavior of everyone who has a reference to the mutable object.

As a symptom of this non-local contract phenomenon, consider the Java collections classes, which are normally documented with very clear contracts on the client and implementer of a class. Try to find where it documents 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? List? Collection? 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:

```java
/**
 * @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:

```java
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:

```java
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:

```java
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 implementor’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 getMitId("bitdiddle") will not return the full 9-digit number, like “928432033”, but instead the obscured version “*****033”.

**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:

```java
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 it 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:

```java
public static char[] getMitId(String username) throws NoSuchUserException
```
This doesn’t 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?

Here’s a much better spec:

```java
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 they 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.

**Summary**

This lecture took a look at mutable objects. 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 crossbar, indicating that the arrow can’t be moved to point to a different object.

The key design principle in this lecture 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.