Reading 9: Mutability & Immutability
Mutability
Risks of mutation
Aliasing is what makes mutable types risky
Specifications for mutating methods
Iterating over arrays and lists
Mutation undermines an iterator
Mutation and contracts
Useful immutable types
Summary

Reading 9: Mutability & Immutability

Software in 6.005

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

Objectives

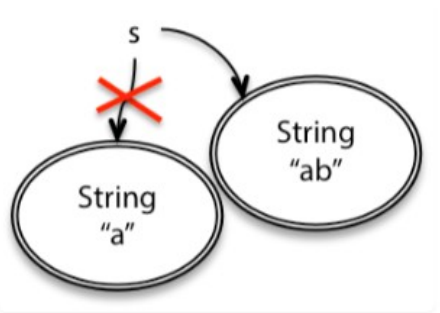
- Understand mutability and mutable objects
- Identify aliasing and understand the dangers of mutability
- Use immutability to improve correctness, clarity, & changeability

Mutability

Recall from *Basic Java* when we discussed snapshot diagrams 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.

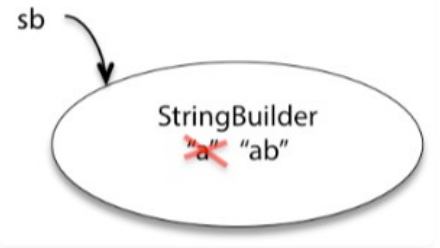
`String` is an example of an immutable type. A `String` object always represents the same string. `StringBuilder` is an example of a mutable type. It has methods to delete parts of the string, insert or replace characters, etc.

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:



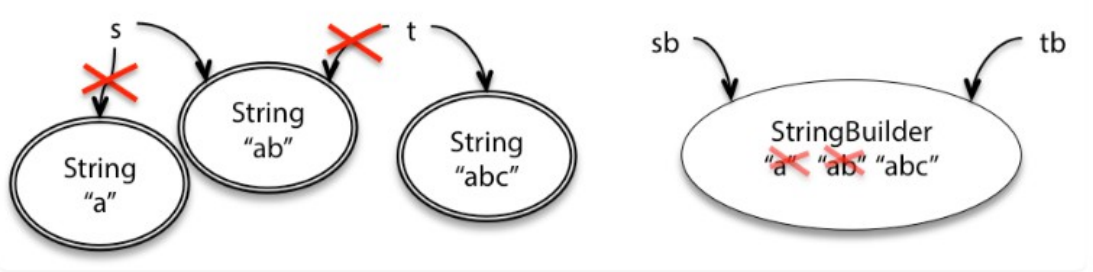

```
StringBuilder sb = new StringBuilder("a");
sb.append("b");
```

`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";

String
Builde
r tb =
sb;
tb.app
end
("c");
```



Why do we need the mutable `StringBuilder` in programming? A common use for it is to concatenate a large number of strings together. Consider this code:

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

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

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

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

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

Risks of mutation

Mutable types seem much more powerful than immutable types. If you were shopping in the Datatype Supermarket, and you had to choose between a boring immutable `String` and a super-powerful-do-anything mutable `StringBuilder`, why on earth would you choose the immutable one? `StringBuilder` should be able to do everything that `String` can do, plus `set()` and `append()` and everything else.

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

Risky example #1: passing mutable values

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

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

Suppose we also need a method that sums the absolute values. Following good DRY practice (Don't Repeat Yourself), the implementer writes a method that uses `sum()`:

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

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

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

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

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

Let's think about the key points here:

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

Risky example #2: returning mutable values

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

Let's consider `Date` , one of the built-in Java classes. `Date` happens to be a mutable type. Suppose we write a method that determines the first day of spring:

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

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

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

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

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

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

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

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

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

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

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

Key points:

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

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

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

(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`.

Before we go any further:

> You should already have read: **Classes and Objects** in the Java Tutorials.

> Read: **the `final` keyword** on CodeGuru.

`MyIterator`

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

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

    private final ArrayList<String> 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;
    }
}
```

`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 , 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` ?

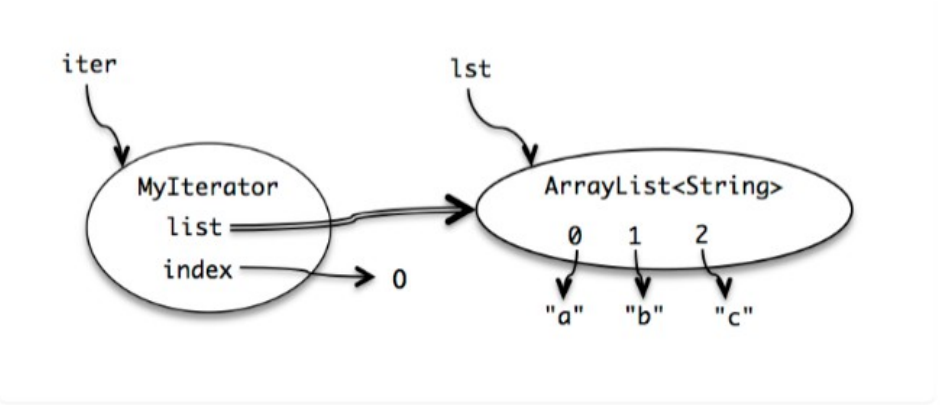
The `static` keyword is missing from `MyIterator` 's methods, which means they are **instance methods** that must be called on an instance of the object, e.g. `iter.next()` .

The `this` **keyword** is used at one point to refer to the **instance object** , in particular to refer to an instance variable (`this.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).

`final` is used to indicate which of the object's internal variables can be reassigned and which can't. `index` is allowed to change (`next()` updates it as it steps through the list), but `list` cannot (the iterator has to keep pointing at the same list for its entire life — if you want to iterate through another list, you're expected to create another iterator object).

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



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

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

Mutation undermines an iterator

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

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

Note that `dropCourse6` has a frame condition (the *modifies* clause) in its contract, warning the client that its list argument will be mutated.

Next, following test-first programming, we devise a testing strategy that partitions the input space, and choose test cases to cover that partition:

```
// Testing strategy:
//   subjects.size: 0, 1, n
//   contents: no 6.xx, one 6.xx, all 6.xx
//   position: 6.xx at start, 6.xx in middle, 6.xx at end

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

Finally, we implement it:

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

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

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

We got the wrong answer: `dropCourse6` left a course behind in the list! Why? Trace through what happens. It will help to use a snapshot diagram showing the `MyIterator` object and the `ArrayList` object and update it while you work through the code.

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

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

then you'll get a `ConcurrentModificationException`. The built-in iterator detects that you're changing the list under its feet, and cries foul. (How do you think it does that?)

How can you fix this problem? One way is to use the `remove()` method of `Iterator`, so that the iterator adjusts its index appropriately:

```
Iterator iter = subjects.iterator();
while (iter.hasNext()) {
    String subject = iter.next();
    if (subject.startsWith("6.")) {
        iter.remove();
    }
}
```

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

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

Mutation and contracts

Mutable objects can make simple contracts very complex

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

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

As a symptom of this non-local contract phenomenon, consider the Java collections classes, which are normally documented with very clear contracts on the client and implementer of a class. Try to find where it documents the crucial requirement on the client that we've just discovered — that you can't modify a collection while you're iterating over it. Who takes responsibility for it? `Iterator` ? `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:

```
/**
 * @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 datab
ase
 */
public static char[] getMitId(String username) throws NoSuchUserExceptio
n {
    // ... 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 NoSuchUserExceptio
n {
    // 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 user
           or throws NoSuchUserException if nobody with username is in M
           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 us
           or throws NoSuchUserException if nobody with username is in M
           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 datal
```

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` and `BigDecimal` are immutable.
- Don't use mutable `Date` s, use the appropriate immutable type from `java.time` 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` 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 `UnsupportedOperation-Exception` .

  Before we pass a mutable collection to another part of our program, we can wrap it in an unmodifiable wrapper. We should be careful at that point to forget our reference to the mutable collection, lest we accidentally mutate it. (One way to do that is to let it go out of scope.) Just as a mutable object behind a `final` reference can be mutated, the mutable collection inside an unmodifiable wrapper can still be modified by someone with a reference to it, defeating the wrapper.
- `Collections` also provides methods for obtaining immutable empty collections: `Collections.emptyList` , etc. Nothing's worse than discovering your *definitely very empty* list is suddenly *definitely not empty* !

Summary

In this reading, we saw that mutability is useful for performance and convenience, but it also creates risks of bugs by requiring the code that uses the objects to be well-behaved on a global level, greatly complicating the reasoning and testing we have to do to be confident in its correctness.

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

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

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