L6: Specifications Part 1

Today
- Preconditions & postconditions
- Tests for a spec
- Exceptions

Required reading (from the Java Tutorial)
Make sure that you read and understand these parts of Java syntax and semantics:

- **Exceptions** (~14 pages)
- **Packages** (6 pages)
- **Controlling Access** (1 page)
  [http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html](http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html)
- **Javadoc Comments** (1 page)

Introduction

In this lecture, we’ll look at the role played by specifications of methods. Specifications are the linchpin of teamwork. It’s impossible to delegate responsibility for implementing a method without a specification. The specification acts as a contract: the implementer is responsible for meeting the contract, and a client that uses the method can rely on the contract. In fact, we’ll see that like real legal contracts, specifications place demands on both parties: when the specification has a pre-condition, the client has responsibilities too.

In today’s lecture, Part 1, we’ll talk about what preconditions and postconditions are, and what they mean for the implementor and the client of a method. We’ll also talk about how to use exceptions, an important language feature found in Java, Python, and many other modern languages, which allows us to make a method’s interface safer from bugs and easier to understand.

Why Specifications?

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

Specifications are good for the client of a method because they spare the task of reading code. If you’re not convinced that reading a spec is easier than reading code, take a look at some of the standard Java specs and compare them to the source code that implements them. ArrayList, for example, in the package java.util, has a very simple spec but its code is not at all simple.

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

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

**Behavioral Equivalence**

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

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

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

Of course the code is different, so in that sense they are different. Our question though is whether
one could substitute one implementation for the other. Not only do these methods have different
code; they actually have different behavior:

- when `val` is missing, `findA` returns the length and `findB` returns -1;
- when `val` appears twice, `findA` returns the lower index and `findB` returns the higher.

But when `val` occurs at exactly one index of the array, the two methods behave the same. It may be
that clients never rely on the behavior in the other cases. So the notion of equivalence is in the eye of
the beholder, that is, the client. In order to make it possible to substitute one implementation for
another, and to know when this is acceptable, we need a specification that states exactly what the
client depends on.

In this case, our specification might be

```plaintext
requires: val occurs in a
effects: returns index i such that a[i] = val
```

**Specification Structure**

A specification of a method consists of several clauses:

- a precondition, indicated by the keyword `requires`;
- a postcondition, indicated by the keyword `effects`.

The precondition is an obligation on the client (i.e., the caller of the method). It’s a condition over
the state in which the method is invoked. If the precondition does not hold, the implementation of
the method is free to do anything (including not terminating, throwing an exception, returning arbitrary results, making arbitrary modifications, etc.).

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

Specifications in Java

Some languages (notably Eiffel) incorporate preconditions and postconditions as a fundamental part of the language, as expressions that the runtime system (or possibly even the compiler) can automatically check to enforce the contracts between clients and implementers.

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

Java has a convention for documentation comments, in which parameters are described by @param clauses and results are described by @return and @throws clauses. You should put the preconditions into @param where possible, and postconditions into @return and @throws. So a specification like this:

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

might be rendered in Java like this:

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

What A Spec May Talk About

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

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

Testing and specifications

In testing, we talk about black box tests that are chosen with only the specification in mind, and glass box tests that are chosen with knowledge of the actual implementation. But it’s important to note that even glass box tests must follow the specification. Your implementation may provide stronger guarantees than the specification calls for, or it may have specific behavior where the specification is undefined. But your test cases should not count on that behavior. Test cases must obey the contract, just like every other client.

For example, suppose you are testing this specification of find:
static int find (int[] a, int val)
  requires: val occurs in a
  effects: returns index i such that a[i] = val

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

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

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

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

**Specification for a Mutating Method**

Our specifications of find didn’t give us the opportunity to illustrate how to describe side-effects in the postcondition.

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

```
static boolean addAll (List<T> list1, List<T> list2)
  requires: list1 != list2, list1!=null, list2!=null
  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. Finally, look at the precondition. It tells us that the behavior of the method is not constrained if you call it with a null argument, or if you attempt to add the elements of a list to itself. The constraint that arguments not be null is often left implicit in practice, but you can easily imagine why the implementor of the method would want to impose the second 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.

This precondition is also explicit 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. In fact, there are extensions to Java that allow you to specify this directly in the type declaration, e.g., static boolean addAll (@NotNull List<T> list1, @NotNull List<T> list2), where it can be checked automatically at compile time or runtime.

Here is another example of a mutating method:

```
static void sort(List<String> lst)
```
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()
```

Exceptions for Special Results

You've probably already seen some exceptions in your Java programming so far, such as ArrayOutOfBoundsException (thrown when an array index a[i] is outside the valid range for the array) or NullPointerException (thrown when trying to call a method on a null object reference). These exceptions generally indicate bugs in your code, and the information displayed by Java when the exception is thrown can help you find and fix the bug.

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

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

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

```java
Calendar lookup (String name)
```

What should the method do if the birthday book doesn’t have an entry for the person whose name is given? Well, we could return some special date that is not going to be used as a real date. Bad programmers have been doing this for decades; they would return 9/9/99, for example, since it was obvious that no program written in 1960 would still be running at the end of the century.

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

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

and the caller handles the exception with a catch clause. Now there’s no need for any special value, nor the checking associated with it.

Checked and Unchecked Exceptions

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

Some terminology: checked exceptions are called that because they are checked by the compiler:
If a method might throw a checked exception, the possibility must be declared in its signature. `FileNotFoundException` would be a checked exception, and that’s why the signature ends `throws` `FileNotFoundException`.

If a method calls another method that may throw a checked exception, it must either handle it, or declare the exception itself, since if it isn’t caught locally it will be propagated up to callers.

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

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

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

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

### Throwable Hierarchy

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

```
Throwable
\--- Exception
\   \--- RuntimeException
\   Error
```

`Throwable` is the class of objects that can be thrown or caught. `Throwable`’s implementation records a stack trace at the point where the exception was thrown, along with an optional string describing the exception. Any object used in a `throw` or `catch` statement, or declared in the `throws` clause of a method, must be a subclass of `Throwable`.

`Error` is a subclass of `Throwable` that is reserved for errors produced by the Java runtime system, such as `StackOverflowError` and `OutOfMemoryError`. For some reason `AssertionError` also extends `Error`, even though it indicates a bug in user code, not in Java runtime. Errors should be considered unrecoverable, and are generally not caught.
Here’s how Java distinguishes checked and unchecked exceptions:

- **RuntimeException, Error, and their subclasses**, are *unchecked* exceptions. The compiler doesn’t require them to be declared in the throws clause of a method that throws them, and doesn’t require them to be caught or declared by a caller of such a method.

- All other throwables – Throwable, Exception, and all of their subclasses except for those of the RuntimeException and Error lineage -- are *checked* exceptions. The compiler requires these exceptions to be caught or declared when it’s possible for them to be thrown.

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

**Exception Design Considerations**

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

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

This suggests a more refined rule:

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

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

- **Queue.pop()** throws an *unchecked* EmptyQueueException when the queue is empty, because it’s reasonable to expect the caller to avoid this with a call like Queue.size() or Queue.isEmpty().
- **Url.getWebPage()** throws a *checked* IOException when it can’t retrieve the web page, because it’s not easy for the caller to prevent this.
- **int integerSquareRoot(int x)** throws a *checked* NotPerfectSquareException when x has no integral square root, because testing whether x is a perfect square is just as hard as finding the actual square root, so it’s not reasonable to expect the caller to prevent it.

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

**Abuse of Exceptions**

Here’s an example from *Effective Java* by Joshua Bloch (Item 39).
try {
    int i = 0;
    while (true)
        a[i++].f();
} catch (ArrayIndexOutOfBoundsException e) { }

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

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

The exception-based idiom is a misguided attempt to improve performance based on the faulty reasoning that, since the VM checks the bounds of array accesses, the normal loop termination test (i < a.length) is redundant and should be avoided. However, because exceptions in Java are designed for use only under exceptional circumstances, few, if any, JVM implementations attempt to optimize their performance. On a typical machine, the exception-based idiom runs 70 times slower than the standard one when looping from 0 to 99.

The exception-based idiom is also not guaranteed to work. Suppose the computation of f() in the body of the loop contains a bug that results in an out-of-bounds access to some unrelated array. If a reasonable loop idiom were used, the bug would generate an uncaught exception, resulting in immediate thread termination with an appropriate error message. If the exception-based idiom were used, the bug-related exception would be caught and misinterpreted as a normal loop termination.

Summary

A specification acts as a crucial firewall between the implementor of a procedure and its client. It makes separate development possible: the client is free to write code that uses the procedure without seeing its source code, and the implementor is free to write the code that implements the procedure without knowing how it will be used. Preconditions make life hard for the client, but, applied judiciously, are a vital tool in the software designer’s repertoire.

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

- **Safe from bugs.** A good specification clearly documents the mutual assumptions that a client and implementor are relying on. Bugs often come from disagreements at the interfaces, and the presence of a specification reduces that. Using machine-checked language features in your spec, like static typing and exceptions rather than just a human-readable comment, can reduce bugs still more.

- **Easy to understand.** A short, simple spec is easier to understand than the implementation itself, and saves other people from having to read the code.

- **Ready for change.** Specs establish contracts between different parts of your code, allowing those parts to change independently as long as they continue to satisfy the requirements of the contract.