

Exceptions and Assertions

6.170 Lecture 11

Spring 2004

In this lecture, we'll look at ways to make programs more reliable using runtime assertions. We'll also discuss Java's exception mechanism, which allows a method to return exceptional results to its caller. Java's exception mechanism is unique in that it provides both checked and unchecked exceptions. We'll look at the difference and consider how to decide when to use a checked exception and when to use an unchecked one.

10.1 Defensive Programming

Let's begin with a simple example:

```
public double sqrt (double x)
// requires: x >= 0
// returns: approximation to square root of x { ... }
```

Now suppose somebody calls `sqrt` with a negative argument. What's the best behavior for `sqrt`? Since the caller has failed to satisfy the precondition, `sqrt` is no longer bound by the terms of its contract, so it is technically free to do whatever it wants: return an arbitrary value, or enter an infinite loop, or melt down the CPU. Since the failed precondition indicates a bug in the caller, however, the most useful behavior would point out the bug as early as possible. We do this by inserting a *runtime assertion* that tests the precondition. Here is one way we might write the assertion:

```
public double sqrt (double x)
// requires: x >= 0
// returns: approximation to square root of x
{
    if (! (x >= 0))
        throw new AssertionError ();
    ...
}
```

When the precondition is not satisfied, this code terminates the program by throwing an `AssertionError` exception. The effects of the caller's bug are prevented from propagating.

Checking preconditions is an example of *defensive programming*. Real programs are rarely bug-free. Defensive programming offers a way to mitigate the effects of bugs even if you don't know where they are.

It is common practice to define a procedure for assertion-checking, usually called `assert`:

```
assert (x >= 0);
```

This approach hides what exactly happens when the assertion fails.

Assertions have the added benefit of documenting an assumption about the state of the program at that point. To somebody reading your code, `assert(x>=0)` says "at this point, it should always be true that $x \geq 0$." Unlike a comment, however, an assertion is executable code that enforces the assumption at runtime.

10.2 Uses for Assertions

Preconditions are only one kind of invariant that can be checked with an assertion. Here are some other invariants you should check with assertions.

Postconditions. Assertions are useful for checking the result of a method before returning it. This kind of assertion is sometimes called a *self check*. For example, the `sqrt` method might square its result to check whether it is reasonably close to `x`:

```
public double sqrt (double x)
// requires: x >= 0
// returns: approximation to square root of x
{
    assert (x >= 0);
    double r;
    ... // compute result r
    assert ( Math.abs(r*r - x) < .0001 );
    return r;
}
```

Rep invariants. The rep invariant of an abstract data type should be asserted whenever an object changes, in order to prove that the invariant is still established. This is easy to do when you implement the rep invariant as a method `checkRep` that tests properties of the rep invariant using assertions.

Checking the rep invariant is much more powerful than checking most other kinds of invariants, because a broken rep often only results in a problem long after it was broken. With `checkRep`, you are likely to catch the error much closer to its source. You should probably assert the rep invariant at the start and end of every method, in case there is a rep exposure that causes the rep to be broken between calls. You should instrument constructors, producers, and mutators, as well as observers that may mutate the rep (as a benevolent side-effect, such as rebalancing a tree or caching a computed value). Ordinary observers can also be instrumented at the start of the method to check for rep exposure.

Covering all cases. If a conditional statement or switch does not cover all the possible cases, it is good practice to use an assertion to block the illegal cases:

```
switch (cardSuit) {
    case CLUBS:    return "clubs";
    case DIAMONDS: return "diamonds";
    case HEARTS:   return "hearts";
    case SPADES:   return "spades";
    default:       assert(false);
}
```

The assertion in the default clause has the effect of asserting that `cardSuit` must be one of `CLUBS`, `DIAMONDS`, `HEARTS`, or `SPADES`.

When should you write runtime assertions? As you write the code, not after the fact. When you're writing the code,

you have the invariants in mind. If you postpone writing assertions, you're less likely to do it, and you're liable to omit some important invariants.

10.3 What *Not* to Assert

Runtime assertions are not free. They can clutter the code, so they must be used judiciously. Avoid trivial assertions, just as you would avoid uninformative comments. For example:

```
// don't do this
x = y + 1;
assert (x == y+1);
```

This assertion doesn't find bugs in your code. It finds bugs in the compiler or Java virtual machine, which are components that you should trust until you have good reason to doubt them. If an assertion is obvious from its local context, leave it out.

Never use assertions to test conditions that are external to your program, such as the existence of files, the availability of the network, or the correctness of input given by the user. Assertions test the internal state of your program to ensure that it is within the bounds of its specification. When an assertion fails, it indicates that the program has run off the rails in some sense, into a state in which that it was not designed to function properly. Assertion failures therefore indicate bugs. External failures are not bugs, and there is no change you can make to your program in advance that will prevent them from happening. External failures should be handled using *exceptions*, which are discussed in more detail later in this lecture.

Many assertion mechanisms are designed so that assertions are executed only during testing and debugging, and turned off when the program is released to users. Java's `assert` statement behaves this way. The advantage of this approach is that you can write very expensive assertions that would otherwise seriously degrade the performance of your program. For example, a procedure that searches an array using binary search has a precondition that the array be sorted. Asserting this precondition requires scanning through the entire array, however, turning an operation that should run in logarithmic time into one that takes linear time. You should be willing (eager!) to pay this cost during testing, since it makes debugging much easier, but not when the program is released to users.

However, disabling assertions in release has a serious disadvantage. With assertions disabled, a program has far less error checking when it needs it most. Novice programmers are usually much more concerned about the performance impact of assertions than they should be. Most assertions are cheap, so they should not be disabled in the official release.

Since assertions may be disabled, the correctness of your program should never depend on whether or not the assertion expressions are executed. In particular, asserted expressions should not have side-effects. For example, if you want to assert that an element removed from a list was actually found in the list, don't write it like this:

```
// don't do this
assert (list.remove (x));
```

If assertions are disabled, the entire expression is skipped, and `x` is never removed from the list. Write it like this instead:

```
boolean found = list.remove (x);
assert (found);
```

10.4 Assertions in Java

In the most recent version of Java (version 1.4), runtime assertions have become a built-in feature of the language. The simplest form of the assert statement takes a boolean expression, exactly as shown above, and throws `AssertionError` if the boolean expression evaluates to false:

```
assert x >= 0;
```

An assert statement may also include a description expression, which is usually a string, but may also be a primitive type or a reference to an object. The description is printed in an error message when the assertion fails, so it can be used to provide additional details to the programmer about the cause of the failure. The description follows the asserted expression, separated by a colon. For example:

```
assert (x >= 0) : "x is " + x;
```

If $x = -1$, then this assertion fails with the error message

```
x is -1
```

along with a stack trace that tells you where the assert statement was found in your code and the sequence of calls that arrived there. This information is often enough to get started in finding the bug.

The assert statement is a recent addition to the Java language, so it is still a little painful to use. First, assert is not recognized by the Java compiler at all unless you add a special switch to the compiler command line:

```
javac -source 1.4 MyClass.java
```

(The equivalent of this switch in Eclipse can be found in `Window/Preferences/Java/Compiler/Compliance`).

A second problem with Java assertions is that assertions are *off by default*. If you just run your program as usual, none of your assertions will be checked! You have to enable assertions explicitly by passing `-ea` (which stands for "enable assertions") to the Java virtual machine:

```
java -ea MyClass
```

(In Eclipse, you specify `-ea` in the Run dialog under VM arguments.)

A third problem is that Java assertions are not backward-compatible. You can't run a compiled program that uses assert statements on a Java virtual machine version 1.3 or earlier.

If you don't like Java's assert mechanism, it's easy enough to roll your own, by writing a procedure that tests a boolean expression and throws an exception if the expression is false. Since assert is now a reserved word in Java, however, you must name your procedure something else:

```
public static void assertAlways (boolean b) {
    if (!b)
        throw new RuntimeException ("assertion failure");
}
```

In fact, the `assertTrue` method in JUnit is basically implemented this way. Since it's a static method, you don't have to be writing a unit test to use `assertTrue`. Just call `Assert.assertTrue()`.

Here's a handy rule for when to use an assertion that's always enabled (like `assertAlways` or `assertTrue`), and when to use an assertion that is disabled in production code:

- **Use assert only for expensive assertions.** Here, *expensive* depends on the context. Checking whether an array is sorted would be expensive in the context of a log-time binary search method; but it would be quite cheap in the context of a method that writes the array to disk.
- **Otherwise use `assertAlways` or `assertTrue`.** Constant-time checks should always fall in this category, unless the constant factor is large or the code is performance-critical.

For more information about Java's assertion mechanism, see

Programming with Assertions
<http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html>

10.5 Responding to Failure

Now we come to a question which we have put off: what should the program do when an assertion fails? Until now, we have assumed that a failure simply aborts execution, but there are other options as well.

You might feel tempted to try and fix the problem on the fly. This is almost always the wrong thing to do. You're unlikely to be able to guess the cause of the failure. If you could, you probably could have prevented the bug in the first place.

On the other hand, it often makes sense to execute some special actions regardless of the exact cause of failure. You might log an error message into a file, or notify the user on the screen. In a safety critical system, deciding what actions are to be performed on failure is tricky and very important. In a radiotherapy machine, for example, you probably want to turn off the radiation source if you detect that something is not quite right.

Sometimes, it's best not to abort execution at all – or at least not right away. When a compiler fails, it makes sense to abort completely. But when a word processor fails, it makes more sense to give the user a chance to save unfinished work if possible, in order to mitigate the effects of the failure.

10.6 Exceptions

How do we structure a program so that we can vary its response to failure? We use *exceptions*.

An exception is a non-local jump. When an exception is thrown, control is passed to the nearest enclosing exception handler — the catch clause of a try-catch statement — that is compatible with the exception. Unlike `break` or `continue`, which can only jump to a label in the current method, an exception can cause control to pass out of the current method to a procedure higher on the call stack.

Exceptions can be used to eliminate preconditions from a method. Consider the `sqrt` method shown earlier:

```
public double sqrt (double x)
// requires: x >= 0
// returns: approximation to square root of x
{
    ...
}
```

The method's behavior is specified only for nonnegative values of x , so it is a partial function. We saw earlier how to use an assertion to make the method more robust, but a caller cannot depend on this behavior, since it is not mentioned in the specification. A better approach, whenever possible, is to make the method a total function, whose behavior is specified for all possible inputs:

```
public double sqrt (double x) throws IllegalArgumentException
// throws: IllegalArgumentException if x < 0
// returns: approximation to square root of x
{
    if (x < 0)
        throw new IllegalArgumentException ();
    ...
}
```

Now the caller can depend on certain behavior when an invalid value is passed. In particular, a caller can catch the exception:

```
try {
    y = sqrt (-1);
} catch (IllegalArgumentException e) {
    e.printStackTrace ();
}
```

This code invokes `sqrt` with an invalid argument, which causes `sqrt` to throw `IllegalArgumentException`. This causes the execution of `sqrt` to abort, without returning a value, and unwinds the call stack until it finds the handler for the exception. Then control is transferred to the handler, and its code is executed.

The `throw` statement doesn't just transfer control. It passes along an object too. This object allows information about the failure to be passed. For example, you can construct an exception with a message string that describes the failure in more detail. Another useful piece of information is the stack trace, which describes where the failure occurred in the code, along with the sequence of method calls that caused the failure. The `printStackTrace` method prints this trace on the standard error stream. The stack trace for this example might look something like this:

```
java.lang.IllegalArgumentException
    at sqrt(Sqrt.java:13)
    at testSqrt(SqrtTest.java:51)
    ...
    at main(TestRunner.java:29)
```

Stack traces should be read from the top down. The first line describes the the exception that was thrown. The second line gives the location of the `throw` statement, in the `sqrt` method located in the source file `Sqrt.java`, line 13. The call to `sqrt` was made by a method called `testSqrt`, in `SqrtTest.java`, line 51, and so forth. The last line on the call stack is the original entry point into the program, which is usually `main`.

An exception keeps unwinding the call stack until it finds a matching handler. As a result, a method can simply pass the exception on instead of catching it:

```
double solveQuad (double a, double b, double c) throws IllegalArgumentException
// returns: x such that  $ax^2 + bx + c = 0$ 
// throws: IllegalArgumentException if no real soln exists
{
    return (-b + sqrt (b*b - 4*a*c)) / (2*a);
}
```

If solveQuad is called with arguments that cause a negative argument to be passed to sqrt, then sqrt throws an exception which is simply reflected unchanged to solveQuad's caller.

What happens if an exception is never caught? If the entire call stack is unwound without finding a handler for the exception, then the program is terminated and the stack trace is printed. (In a multithreaded program, however, only the current thread is terminated by an uncaught exception.)

10.7 Exceptions for Special Results

There's something a little funny about throwing an IllegalArgumentException when solveQuad fails to find a solution. With sqrt, the exception made sense, because it's easy for the caller to check and avoid passing a negative argument. But the caller can't tell how to avoid calling solveQuad when it might have no real solutions. So instead of blaming the caller for passing "illegal" arguments, let's introduce a new exception:

```
double solveQuad (double a, double b, double c) throws NotRealException
// returns: x such that ax^2 + bx + c = 0
// throws: NotRealException if no real solution exists
```

This point is deeper than simply renaming the exception. It's a new way to regard an exception, as returning a special result from the computation. From this perspective, the solveQuad method does some work and returns one of two results: either a real solution if one exists, or NotRealException if no real solution exists. NotRealException lets us indicate when the method cannot return a result in the normal range of its return value.

With this new specification, the body of quadSolve might look like this:

```
double solveQuad (double a, double b, double c) throws NotRealException
// returns: x such that ax^2 + bx + c = 0
// throws: NotRealException if no real solution exists
{
    try {
        return (-b + sqrt (b*b - 4*a*c)) / (2*a);
    } catch (IllegalArgumentException e) {
        throw new NotRealException ();
    }
}
```

The catch clauses catches the IllegalArgumentException thrown by sqrt and throws NotRealException instead. This is called exception translation: converting an exception that makes sense only in the method's implementation into a different exception that is appropriate to the method's interface.

We also have to define the new exception as a class:

```
class NotRealException extends Exception {
    NotRealException () { super (); }
    NotRealException (String msg) { super (msg); }
}
```

All exceptions may have a string message associated with them, which is why NotRealException has two constructors. If not provided in the constructor, the message is null. An exception is a full-fledged object, however, so you are free to add additional fields and methods to an exception which provide more information about the failure or special result.

For example, `NotRealException` might be augmented with methods to obtain the coefficients of a complex solution to the quadratic equation.

Another way to handle special results is to return special values. Many operations in the Java library are designed like this. For example, `List.indexOf()` searches for an object in the list and returns its index if the object is found, or `-1` if the object is not found. When the Java library method `Math.sqrt()` is passed a negative argument, it returns a special floating-point value, `NaN` ("Not a Number").

This approach is OK if used sparingly. Before you consider using a special return value, however, you should ask yourself two questions:

1. **Can the special return value always be distinguished from a normal return?** `List.indexOf()` can safely return `-1` when the object is not found because `-1` can never be a valid index. But another Java library method is not so lucky: `Map.get()` returns `null` to indicate that the key is not found in the map, but this is indistinguishable from the case where the key is found in the map but bound to `null`.
2. **What happens if the caller forgets to check for the special return value?** In the best case, the special return value should fail if the caller attempts to use it as if it were a normal result. For example, using a `null` reference usually causes a `NullPointerException`. The worst case is when an unchecked special return value quietly contaminates the computation, with no obvious failure.

Exceptions have neither of these problems. For the first problem, exceptional returns are syntactically distinguished from normal returns, because an exception must be handled by a catch clause. For the second, Java gives you checked exceptions.

10.8 Checked and Unchecked Exceptions

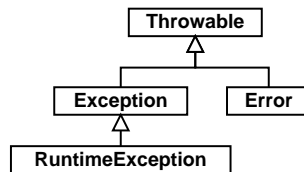
We've seen two different purposes for exceptions: failures and special results. Java provides two different kinds of exception for these two purposes. They behave the same at runtime; the only difference is what kind of checking the compiler provides.

A checked exception is a subclass of `Exception`. If a method might throw a checked exception, the possibility must be declared in its signature. `NotRealException` would be a checked exception, which is why the signature for `quadSolve` declares it in its throw clause. 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.

So if you call `quadSolve` and forget to handle the exception, the compiler will reject your code. This is very useful, because it ensures that exceptions that are expected to occur should be handled. On the other hand, exceptions that correspond to failures are not expected to be handled except at the top level, and for reasons of modularity, we wouldn't want to declare the possibility of failure at every level.

An unchecked exception is a subclass of `RuntimeException`. `NullPointerException` and `IllegalArgumentException` are examples. For an unchecked exception, in contrast, the compiler will not check for catch clauses or throws declarations. Java allows you to write a throws clause for an unchecked exception, such as `throws NullPointerException`, but this has no effect; the compiler doesn't care. When you write a method specification, however, you should document the unchecked exceptions that the method might throw itself, as we did for `sqrt` above. This list is not truly complete, because you cannot anticipate what unchecked exceptions might be propagated from code that is called by your method. However, it is sufficient for specification, since an unexpected exception indicates that your method (or some code it depends on) is broken, and hence is no longer satisfying its specification anyway.

The complete class hierarchy for Java exceptions is shown below.



- Throwable is the class of objects that can be thrown or caught. Any object used in a throw or catch statement, or declared in the throws clause of a method, must be a subclass of Throwable. Throwable’s implementation records a stack trace at the point where the exception was thrown, along with an optional string describing the exception.
- 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 failure in user code, not in the Java runtime. Errors should be considered unrecoverable, and are generally not caught.
- Errors and RuntimeExceptions are unchecked. All other Throwables and Exceptions are checked.

It’s somewhat unfortunate that Java has no typographical convention that distinguishes between checked and unchecked exceptions: when you’re reading code, you can’t easily tell whether you *have* to catch `MalformedURLException` or `ArrayIndexOutOfBoundsException`. (For the former, yes; for the latter, no.) This problem particularly bites when you’re reading a specification of a method. Bloch (p. 181) suggests a useful documentation convention, which we’ll generally adhere to in this class: checked exceptions should be declared in the method signature, but unchecked exceptions should *not*. Both kinds should still be documented in `@throws` clauses, however. For example:

```

public double sqrt(double x)
// returns: approximation to square root of x
// throws: IllegalArgumentException if x < 0

double solveQuad(double a, double b, double c) throws NotRealException
// returns: x such that ax^2 + bx + c = 0
// throws: NotRealException if no real solution exists
  
```

Notice that `sqrt`’s signature omits mentioning `IllegalArgumentException`, because it’s an unchecked exception that doesn’t need to be caught by the client.

10.9 Design Considerations

The rule we have given – use checked exceptions for special results, and unchecked exceptions for failures – makes sense, but it isn’t the end of the story. The snag is that exceptions in Java aren’t as lightweight as we’d like them to be.

Aside from the performance penalty, exceptions in Java incur another (more serious) cost: they’re a pain to use. If you design a method to have its own 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 can prove to yourself that the exception will never be thrown. This latter stipulation creates a dilemma. Suppose, for example, you’re designing a queue abstraction. Suppose you want to support a style of programming in which the queue is popped 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 (see Liskov, p. 73):

- Use an unchecked exception only if you expect that clients will usually write code that ensures the exception will not happen; either because there is a convenient and inexpensive way to avoid the exception, or because the exception reflects unexpected failures.
- Otherwise use a checked exception.

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