Assertions and Exceptions

6.170 Lecture 11

Fall 2005

10.1. Introduction

In this lecture, we'll look at Java's exception mechanism. As always, we'll focus more on design issues than the details of the language, which can be found in one of the recommended Java texts. We'll discuss the general issue of making your code more robust with runtime assertions; how to use exceptions for communicating back to callers of a method, even in non-failure situations; and how to decide whether to use checked or unchecked exceptions.

10.2. Defensive Programming

Suppose you are designing a compiler. Since the compiler is under constant development, and is large and complex, it is likely to contain some bugs despite your best efforts. *Defensive programming* is a way to mitigate the effects of bugs without knowing where they are. Adopting defensive strategies is not an admission of incompetence. Although any large program is likely to have bugs in it, even a bug-free program can benefit from defensive programming. You may have to deal with a database that may occasionally be corrupted. It would be too hard to predict every possible form of corruption, so you take a defensive strategy instead.

Here's how defensive programming works. When you're writing some code, you figure out conditions that you expect to hold at certain points in the code; these are called *invariants*. Then, rather than just assuming that these conditions hold, you test for them explicitly. If a condition is false, you abort the computation.

For example, suppose that in your compiler you maintain a vector of symbols, and some other vectors, each of which contains objects holding information about symbols.

```
class Symbols {
   Vector symbols;
   Vector types;
   Vector positions;
       ...
}
```

To access the information for a given symbol, you have to access the element of the information vector with the same index. Given a symbol `sym`, to obtain its type for example, we take the element of the `types` vector in the same position:

```
int i = symbols.indexOf (sym);
Type t = types.get (i);
```

This code will only work if the symbol being looked up is indeed in the `symbols` vector, and if the two vectors have the same length, so that the indexing in the second statement succeeds. If the second statement fails, then something has gone wrong, and it is unlikely that the compiler will be able to generate correct code. So it would be best to waste no more resources and terminate the program immediately. In fact, we can do better and notice the problem after the first statement.

We do this by inserting a *runtime assertion:*

```
int i = symbols.indexOf (sym);
if (! (i >= 0))
  System.exit (1);

...
Type t = types.get (i);
```

If the call to `indexOf` returns `-1`, indicating that the symbol is not in the vector, we terminate the entire compiler by calling the special method `exit` of the System class.

To make runtime assertions easy to write, most programmers define a procedure, so they can write, for example:

```
assert (i >= 0)
```

This also demonstrates the documentation value of assertions. Even if they are not executed, they help someone reading the code immeasurably to understand what it's doing.

Two important questions arise:

- Where and what kind of runtime assertions are best?
- How should you abort execution?

10.3. Runtime Assertions

First, runtime assertions shouldn't be used as a crutch for bad coding. You want to make your code bug-free in the most effective way. Defensive programming doesn't mean writing lousy code and peppering it with assertions. If you don't already know it, you'll find that in the long run it's much less work to write good code from the start; bad code is often such a mess it can't even be fixed without starting over again.

When should you write runtime assertions? As you write the code, not later. When you're writing the code you have invariants in mind anyway, and writing them down is a useful form of documentation. If you postpone it, you're less likely to do it.

Runtime assertions are not free. They can clutter the code, so they must be used judiciously. Obviously you want to write the assertions that are most likely to catch bugs. Good programmers will typically use assertions in these ways:

- At the start of a procedure, to check that the state in which the procedure is invoked is as expected. This makes sense because a high proportion of errors are related to misunderstandings about interfaces between procedures.
- At the end of a complicated procedure, to check that the result is plausible. This kind of assertion is sometimes called a *self check*.
- When an operation is about to be performed that has some external effect.

Runtime assertions can also slow execution down. Novices are usually much more concerned about this than they should be. Do not use runtime assertions for testing the code and turn them off in the official release. A good rule of thumb is that if you think a

runtime assertion is necessary, you should worry about the performance cost only when you have evidence (eg, from a profiler) that the cost is really significant.

Nevertheless, it makes no sense to write absurdly expensive assertions. Suppose, for example, you are given an array and an index at which an element has been placed. It would be reasonable to check that the element is there. But it would not be reasonable to check that the element is nowhere else, by searching the array from end to end: that might turn an operation that executes in constant time into one that takes linear time (in the length of the array).

10.4. Assertions in Java

In 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 and throws `AssertionError` if the boolean expression evaluates to false:

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

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
```

A second problem with Java assertions is that assertions are off by default. If you just run your program, 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
```

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 1.3 or earlier. For more information on programming with assertions, see http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html .

10.5. Responding to Failure

Now we come to the question of what to do when an assertion fails.

You might feel tempted to try and fix the problem on the fly. This is almost always the wrong thing to do. It makes the code more complicated, and usually introduces even more bugs. You’re unlikely to be able to guess the cause of the failure; if you can, you could probably have avoided the bug in the first place.

Depending on the type of program, it may make sense to abort completely, or not abort. On the other hand, it often makes sense to execute some special actions irrespective of the exact cause of failure. You might log the failure to a file, and/or notify the user on the screen, for example.

How do you structure an abort of a command in the code? With the mechanisms we’ve seen so far, it’s extremely tedious. You’d need to give each procedure a special return value that indicates whether it succeeded or failed, then at every call you’d need to check this value. When a procedure call fails, the calling procedure must then itself return a failure value. Your code will look something like this:

```
boolean p (...) {
  boolean success = q ( ...);
  if (!success) return false;
  success = r (...);
  if (! success) return false;
    ...
}
```

Aside from being tedious (and depriving you of additional return values -- a big problem in a language that only allows one return value), this approach has another problem. *What is it?*

10.6. Non-local Jumps

What we need is a mechanism for making a *non-local jump*. Look at this code:

```
class Symbols {
  Vector symbols;
  Vector types;
  Vector positions;
  Type getType (Symbol s) {
    int i = symbols.indexOf (s);
    if (! (i >= 0))
      throw new RuntimeException ("getType");
  }
      ...
}
class Compiler {
  Symbols symbols;
    ...
  void compile () {
    try {
      parse ();
      typecheck ();
      optimize ();
      generate ():
    }
    catch (RuntimeException e) {
      System.out.println ("Failed at: " + e.getMessage ());
      }
  }
  void typecheck () {
      ...
    Type t = symbols.getType (s);
      ...
  }
}
```

This code shows how exceptions can be used in our compiler to abort without killing the entire execution. Even in the compiler this is useful, because it lets us print an informative error message. If the assertion in `getType` fails, an exception is 'thrown'. This not only causes the execution of `getType` to abort, but also the execution of `typeCheck`, its caller. Execution of each caller in the call stack is aborted, until a

method such as `compile` is reached that has a `handler` for the exception. Now control is transferred to the handler, labelled `catch`, and its code is executed.

The throw command doesn't just transfer control: it passes an object along too. This allows information about a failure to be passed. The constructor for the exception takes a message string which is extracted in the handler. Here we've used it to convey information about where in the code the failure occurred. This is a common strategy; you can make it more useful by adding more details, such as the particular symbol that was looked up.

10.7. Exceptions for Special Results

Exceptions are not just for handling failures. 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 with exceptions you can get help from the compiler in this.)

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

```
Date 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:

```
Date 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.

Novice Java programmers often make the mistake of using null references as special values. Be warned that this is a bad idea, and can lead to code riddled with null dereferencing problems.

10.8. Abuse of Exceptions

Here's an example of 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.

10.9. 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.

If a method might throw a *checked* exception, the possibility must be declared in its signature. `NotFoundException` would be a checked exception, and that's why the signature ends `throws NotFoundException`. 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 the `lookup` method 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.

For an *unchecked* exception, in contrast, the compiler will not check for try-catch or a throws declaration. Java still allows you write a throws clause as part of a signature for an unchecked exception, but this has no effect (and is thus a bit funny, and I don't recommend doing it).

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

10.10. Exceptions and Preconditions

An obvious design issue is whether to use a precondition, and if so, whether it should be checked. It is crucial to understand that a precondition does not require that checking be performed. On the contrary, the most common use of preconditions is to demand a property precisely because it would be hard or expensive to check.

As mentioned above, a non-trivial precondition renders the method partial. This inconveniences clients, because they have to ensure that they don't call the method in a bad state (that violates the precondition); if they do, there is no predictable way to recover from the error. So users of methods don't like preconditions, and for this reason the methods of a library will usually be total. That's why the Java API classes, for example, invariably throw exceptions when arguments are inappropriate. It makes the programs in which they are used more robust.

Sometimes though, a precondition allows you to write more efficient code and saves trouble. For example, in an implementation of a binary tree, you might have a private method that balances the tree. Should it handle the case in which the ordering invariant of the tree does not hold? Obviously not, since that would be expensive to check. Inside the class that implements the tree, it's reasonable to assume that the invariant holds. As another example, in the Java standard library, for example, the binary search methods of the `Arrays` class require that the array given be sorted. To check that the array is sorted would defeat the entire purpose of the binary search: to obtain a result in logarithmic and not linear time.

The decision of whether to use a precondition is an engineering judgment. The key factors are the cost of the check (in writing and executing code), and the scope of the method. If it's only called locally in a class, the precondition can be discharged by carefully checking all the sites that call the method. But if the method is public, and used by other developers, it would be less wise to use a precondition.

If you have used a precondition, and you wish to check for it, if the precondition is violated, you should throw an unchecked exception. The client cannot be expected to handle the exception, since he/she did not know enough to set up the method call properly in the first place.

10.11. Design Considerations

The rule we have given -- use checked exceptions for special results, and unchecked exceptions to signal 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. 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 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 or

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.

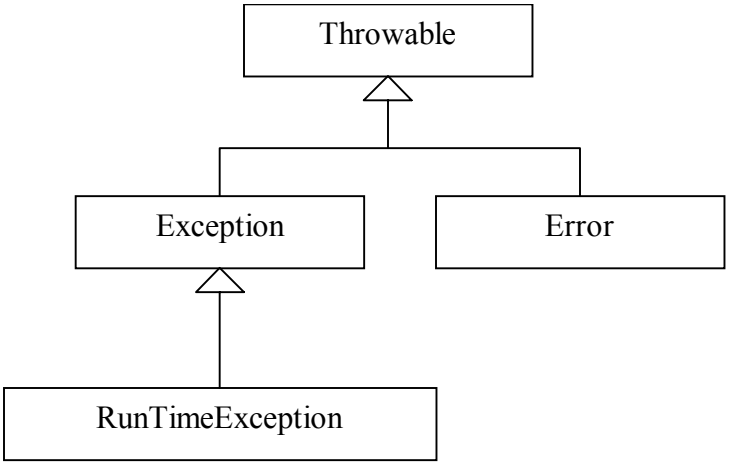
This suggests a more refined rule (see course text, p 73):

- You should use an unchecked exception only 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, or because the exception reflects unexpected failures;
- Otherwise you should use a checked exception.

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.

10.12. Throwable Hierarchy

Let's look at the class hierarchy for Java exceptions.



`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 Java runtime. Errors should be considered recoverable, and are generally not caught. All the unchecked throwables you implement should subclass `RunTimeException` (directly or indirectly).

`The classes Error` and `RuntimeException` correspond to unchecked exceptions. All other `Throwables` and `Exceptions` are checked. Do not define a throwable that is

not a subclass of `Exception`, `RunTimeException`, or `Error`.

10.13. Summary

Today, we've seen how exceptions can provide non-local jumps. These allow failures to be handled gracefully, and support a common style of defensive programming. Exceptions are also often a better way to convey results than special values.We looked at some features specific to Java, in particular the distinction between checked and unchecked exceptions, and how to choose which one to throw.