Object-Oriented Design	Lecture 6
CSU 370 Fall 2008 (Pucella)	Tuesday, Sep 30, 2008

Errors and Exceptions

Consider the kind of errors that can occur in programs, generally speaking. There are various errors possible, and it makes sense to classify them somehow.

(1) Trying to apply an operation that works only on some type of values on vales of the wrong type. For instance, trying add two Boolean values, or trying to divide strings.

(2) Trying to invoke a method `m` on an object that does not define `m`, for instance, if `d` is an instance of `Drawing`, trying to invoke `d.playTune()` is an error.

(3) Invoking an operation with arguments on which the operation is undefined. For instance, dividing by 0, or taking the tangent of $\pi/2$.

(4) Casting an object to a class that it is not an instance of, e.g., without first checking `instanceof`. (More about this later.)

(5) Problems out of a programmer's control, such as hardware failures. For instance, disk failures during disk IO, or network failure during network IO.

These distinctions are not sharply defined; some errors may well be classified in different categories. But the categories are useful as a general sense of the kind of errors that arise.

Different languages check and deal with these errors differently. Let's focus on the first, type errors. Languages may check for type errors at run time, that is, during program execution. These language support *dynamic type checking*. Scheme supports dynamic type checking. Other languages actually can check for type errors at compile time, that is, before programs execute. They support *static type checking*. Java supports static type checking.

There are advantages to checking for type errors at compile time; in particular, you still have a chance to correct the problem while the code is in your hands. With dynamic type checking, errors may only show up after the code has been shipped and the piece of software is in the hands of the customer, making it more difficult and expensive to correct. Static type checking also has some disadvantages. In particular, there is no way to identify exactly all those programs that have type errors at compile time.[1] Thus, the type checker needs

[1] This is a deep limitation in what we can say about programs in general, which you will see in a good theory of computation course. It is a consequence of the so-called *undecidability of the halting problem for Turing machines.* Very roughly speaking, the limitation is that it is impossible to write a program that takes a program P as input and (without executing the program) answers correctly a question about how P will execute. Here, the program that checks a property is the type checker.

to approximate, and it will approximate conservatively. Thus, there are programs that are in fact type correct (that would not cause a problem during execution) but that the type checker will report as having a type error.

So, Java takes care of type errors, errors of type (1) above, at compile time. It also takes care of errors of type (2) at compile-time. The other kind of errors, however, cannot be reliably checked for at compile time. (Here again, the limitation in the footnote above bites us here.) Every language will have a different way to report those kind of errors. Generally, the error will abort execution and report a useful error message on the console. But in many languages, Java included, these errors can be dealt with within the program itself, and the execution need not actually abort. In other words, these errors can often be recovered from gracefully.

The modern approach to dealing with runtime errors, those not taken care of at compile-time, is to use *exceptions*. An exception is just an object in the system, that gets created when an error is encountered, and that propagates through the code until either it is handled, or aborts execution.

There is actually a whole hierarchy of classes in Java implementing exceptions. This hierarchy lets us distinguish the kind of exceptions that can occur. Here is a partial class hierarchy of exceptions:

```
Throwable
 |
 +-- Error
 |    |
 |    +-- OutOfMemoryError
 |    +-- AssertionError
 |
 +-- Exception
      |
      +-- IOException
      +-- InterruptedException
      +-- RuntimeException
           |
           +-- ArithmeticException
           +-- ClassCastException
           +-- NullPointerException
           +-- IllegalArgumentException
           +-- NumberFormatException
```

The class `Throwable` is the most general kind of exception that every other exception subclasses. The `Error` class roughly represent the show-stoppers, that lead to aborting execution in almost all cases. The `Exception` class capture more "benign" forms of errors. These include `IOExceptions`, representing exceptions due to failure of IO (disk failure, network failure, and so on), while `RuntimeException`s represent exceptions such as dividing by 0 (an `ArithmeticException`), casting an object to an unacceptable class (a `ClassCastException`), invoking a method on a null object (a `NullPointerException`). The `IllegalArgumentException` is a general exception to represent passing a wrong value to a method.

(Note that you can also create new kinds of exceptions by subclassing, generally by subclassing the `Exception` class. We'll see subclassing more extensively in the coming weeks.)

Many exceptions are created automatically by the system when an error is encountered. You can also cause an exception yourself in the code. This is called *throwing* an exception:

```
throw new IllegalArgumentException ("Oops, something bad happened");
```

To a first approximation, throwing an exception abort execution of the program.

Here is a simple, possibly naive, but still accurate model that explains how exceptions affect code execution. Intuitively, you can think of the following two rules that apply to all programs:

(1) Every method returns either a value as specified by its signature, or an exception. The statement `return` returns a value from the method, while the statement `throw` returns an exception from the method.

(2) Every method call is implicitly wrapped by code that checks if the method called returned an exception—if so, it returns the exception immediately, using a `throw`, otherwise it just continues execution with the value returned by the method.

In this way, exceptions propagate all the way back to the `main` method. If the `main` method returns an exception, then that exception is reported to the user.

In fact, the fact that exceptions can be returned from methods sometimes has to appear in the signature. Java distinguishes between checked and unchecked exceptions. Unchecked exceptions (including `Error`s and `RuntimeException`s) are exceptions that can occur essentially at any point during execution. Checked exceptions (including `IOException`s) and all exceptions that you will create) are exceptions that can occur only when specific methods throw them. You are required to annotate every method that can throw a checked exception, either because it can throw it directly, or because it invokes a method that can throw that exception.

```
public void someMethod () throws SomeException {
  // some code that can throw the SomeException exception
}
```

This is all good and well, but how can we deal with such exceptions gracefully? After all, if exceptions were just errors that cannot be dealt with, we could just say that an exception aborts execution, and be done with it. The idea is that if a method call appear in the body of a `try` block, and if that call returns an exception, that exception is not returned immediately, but instead it is handled `catch` clause associated with the `try` block.

Suppose `MyException` is a new exception you have defined, subclassing `Exception`, and suppose you wanted to intercept this exception if it is thrown by the method `someMethod()` on object `obj`:

```
try {
  obj.someMethod();
} catch (MyException e) {
  // some code to deal with the exception
}
```

This reads: if a `MyException` is returned by `obj.someMethod()`, then instead of having that exception returned immediately, intercept it, name it `e`, and continue execution of the current method with the code in the `catch` clause. (The reason why we may be interested in `e` is that `e` is an object that actually implements some useful methods such as `getMessage()` which returns a string representing the message associated with the exception.) This lets you gracefull recover from an exception, by intercepting it and dealing with it instead of letting it bubble up until it aborts the entire program with an error.

Note that a `catch` clause catching an exception `SomeException` will in fact catch all exceptions that are instances of subclasses of `SomeException`. This lets you intercept, for instance, any possible exception in a single catch clause:

```
try {
  // some code doing something interesting
} catch (Throwable e) {
  // deal with the exception
}
```

This works because every exception ultimately subclasses `Throwable`. This kind of code is useful in testing code to nicely format the exception and report useful information.

It is also possible to catch multiple exceptions and dealing with them differently:

```
try {
  // some code doing something interesting
} catch (SomeExceptionClass e) {
  // deal with this kind of exception
} catch (SomeOtherExceptionClass e) {
  // deal with this other kind of exception
}
```

Note that the order in which the `catch` clauses occur is relevant—they are tried in order, and the first clause where the current exception is in a subclass of the specified exception will be chosen.