The University of Melbourne
Department of Computer Science and Software Engineering
433-254 Software Design
Semester 2, 2003

Answers for Tutorial 7
Week 8

1. What are exceptions and how are they handled in Java? Give an example.

   Sample Answer:

   If a Java program attempts to carry out an illegal operation at runtime, it does not necessarily halt its processing at that point. In most cases, the Java Virtual Machine provides the possibility of catching the problem and recovering from it. To emphasize that such problems are not fatal, the term *exception* is used rather than *error*.

   Java *exception* handling provides a systematic way for the user to respond to runtime errors and to decide on an appropriate response rather than simply letting the program come to a halt.

   Without having an exception handling system built-in to the language, you would have to write your own routines to validate data and handle all the other possible runtime situations before they cause a halt in your program.

   For example, in the following code we create a special method to test if a `String` holds a valid integer value before the string goes to the `parseInt` method.

```
String str = getParameter("dat");
  if ( testIfInt(str))
    Integer.parseInt(str);
  else
  {
    ErrorFlag = MY_ERROR_BAD_FORMAT;
    return -1;
  }
  boolean testIfInt(String str)
  {
    ... code to test characters for numbers ..
  }
  ...
```

   Fortunately, we can instead use the Java `try{}catch(Exception e){}` operation to handle exceptions. In the following code segment, for example, we surround the `parseInt` method invocation with a `try-catch` pair.

```
try
   {
     int data =
        Integer.parseInt(getParameter("dat"),10);
```

```
    aFunctionSetup(data);
  } catch ( NumberFormatException e)
  {
    data = -1;
  }
```

Thus, if the string passed to `parsInt` does not represent a valid integer number, `parseInt` *throws* an instance of the class `NumberFormatException`, which will cause the program execution to jump to the first line in the `catch` block.

The `parseInt` method in class `Integer` is written something like the following:

```
public static int parseInt(String s, int radix)
    throws NumberFormatException
{
  ...
  ...code to check if the string is a number and if
  ...it isn't then:
  throw new NumberFormatException(s);
  ...
}
```

The `throws NumberFormatException` phrase in the method signature indicates that the method includes a `throw` statement for this exception.

We see that `throw` statement actually creates an instance of the exception and causes the routine to return with the exception thrown. The constructors for an exception may include arguments with which you can pass useful information about the circumstances of the exception. The catch code can then examine this info using methods for the particular type of exception.

Java divides exceptions into two categories:

1. General exceptions
2. Run-time exceptions

The *general* exceptions *must* be handled, i.e. a `try-catch` must either be nested around the call to the method that throws the exception or the method must explicitly indicate with `throws` that it can generate this exception. (Then other methods that invoke this method must catch the exception.) The compiler will throw an error message if it detects an uncaught exception and will not compile the file.

The *run-time* exceptions do not have to be caught. This avoids requiring that a `try-catch` be place around, for example, every integer divide

operation to catch a divide by zero or around every array variable to watch for indices going out of bounds.

However, you should handle possible run-time exceptions if you think there is a reasonable chance of one occurring.

You can use multiple catch clauses to catch the different kinds of exceptions that code can throw as shown in this snippet:

You can use multiple catch clauses to catch the different kinds of exceptions that code can throw as shown in this snippet:

```
try
  {
    ... some code...
  } catch ( NumberFormatException e)
  {
    ...
  } catch (IOException e)
  {
    ...
  } catch (Exception e)
  {
    ...
  } finally // optional
  {
    ...this code always executed even if
       no exceptions...
  }
```

Here there are two `catch` statements for `Exception` subclasses and one for any other `Exception` instance. Regardless of whether the code generates an exception or not, the `finally` code block will be executed.

Exception handling is thus based on `try-catch` operation, the `throws` and `throws` keywords, and the `Exception` class and its subclasses.

2. While reading a file, how would you check whether you reached the end of the file?

   Sample Answer:

   Some methods simply return -1 (e.g. `read`() in class `DataInputStream` or class `FileReader`) if there are no more data because the end of stream has been reached.

Some other methods throw EOFException (e.g. `readFully(byte[] b)` in class `DataInputStream`), if they reach end of file before completing their read.

3. How do we design, create, and access a package? Discuss with suitable example.

   Sample Answer:

   A package is the Java version of a *library*. A package refers simply to a group of related class files in the *same directory*. Each class file, however, includes a `package` directive with that directory name at the top of the file.

   For example, say that we put the files `TestA.java` and `TestB.java` into the directory `myPack`, which is a sub-directory of `myApps`. So on a Windows platform the file path looks like

   ```
   c:\myApps\myPack\TestA.java
   ```

   and

   ```
   c:\myApps\myPack\TestB.java.
   ```

   At the top of each file we put the statement

   ```
   package myPack;
   ```

   as shown in the following code:

myApps/myPack/TestA.java	**myApps/myPack/TestB.java**

`package myPack;`
 	
`public class TestA`
`{`
`  public int a;`
`  public TestA(int arg1)`
`  {`
`  a = arg1;`
`  }`
`}`
	`package myPack;`
	
	`public class TestB`
	`{`
	`  public double x;`
	`  public TestB(double y)`
	`  {`
	`  x = y;`
	`  }`
	`}`

   The `import` directive tells the compiler where to look for the class definitions when it comes upon a class that it cannot find in the default `java.lang` package. A class that wants to potentially use any of the classes from myPack package (i.e. TestA and TestB classes) may use an `import` statement like:

   ```
   import myPack.*;
   ```

But, if the class which is trying to call our TestA and/or TestB classes is not in myApps directory then we need to tell the compiler were to look for myPack package by including its absolute path into CLASSPATH environment variable:

set CLASSPATH=c:\myApps;$CLASSPATH

4. Discuss the various levels of access protection for packages and their implications?

   Sample Answer:

   Basically, access control levels for packages follows the general access control rules in Java, which are summerised in the following table:

Specifier	class	subclass	package	world
`private`	X			
`protected`	X	X*	X	
`public`	X	X	X	X
`package`	X		X	

   The first column indicates whether the class itself has access to the member defined by the access specifier. As you can see, a class always has access to its own members. The second column indicates whether subclasses of the class (regardless of which package they are in) have access to the member. The third column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The fourth column indicates whether all classes have access to the member.

   For more detailed discussion of access control levels and rules, please refer to: http://java.sun.com/docs/books/tutorial/java/javaOO/accesscontrol.html