Introduction to Java™

Module 6: Exceptions

Error Handling

- The basic philosophy of Java is that **"badly-formed code will not be run."**
- As with C++, the ideal time to catch the error is at compile time.
- However, not all errors can be detected at compile time.
- The rest of the problems must be handled at run-time.
- Some formality allows the originator of the error to pass appropriate information to a recipient who will know how to handle the difficulty properly.

Error Handling

- In C and other earlier languages there could be several of these formalities.
- They were generally established by convention and not as part of the programming language.
- Typically, you returned a special value or set a flag, and the recipient was supposed to look at the value or the flag.
- However, programmers who use a library tend to think of themselves as invincible: "Yes, errors might happen to others but not in ***my*** code."
- This approach to handling errors was a major limitation to creating large, robust, maintainable programs.

Error Handling

```
errorCodeType readFile {
        initialize errorCode = 0;
        open the file;
        if (theFileIsOpen) {
            determine the length of the file;
            if (gotTheFileLength) {
                allocate that much memory;
                if (gotEnoughMemory) {
                    read the file into memory;
                    if (readFailed) {
                        errorCode = -1;
                    }
                } else {errorCode = -2;}
            } else {errorCode = -3; }
            close the file;
            if (theFileDidntClose && errorCode == 0) {
                errorCode = -4;
            } else {
                errorCode = errorCode and -4;
            }
        } else {errorCode = -5;}
        return errorCode;
    }
```

```
readFile {
        try {
            open the file;
            determine its size;
            allocate that much memory;
            read the file into memory;
            close the file;
        } catch (fileOpenFailed) {
            doSomething;
        } catch (sizeDeterminationFailed) {
            doSomething;
        } catch (memoryAllocationFailed) {
            doSomething;
        } catch (readFailed) {
            doSomething;
        } catch (fileCloseFailed) {
            doSomething;
        }
    }
```

Error Handling with Exceptions

- Exception handling is enforced by the Java compiler.
- You can generate your own exceptions.
- An *exceptional condition* is a problem that prevents the continuation of the method or scope that you're in.
  - With an exceptional condition, you cannot continue processing.
  - All you can do is jump out of the current context and relegate that problem to a higher context.
  - This is what happens when you throw an exception.

Error Handling with Exceptions

- At the point where the problem occurs:
  - You might not know what to do with it
  - You must stop and somebody, somewhere, must figure out what to do.
  - Don't have enough information in the current context to fix the problem.
  - You hand the problem out to a higher context where someone is qualified to make the proper decision (much like a chain of command).

Exceptions. So What?

- A significant benefit of exceptions is that they clean up error handling code.
- No need of checking for a particular error and dealing with it at multiple places in your program.
- The exception will guarantee that someone catches it.
- Need to handle the problem in only one place, the so-called *exception handler*.
- This saves you code and it separates the code that describes what you want to do from the code that is executed when things go awry.
- In general, reading, writing, and debugging code becomes much clearer with exceptions.

Exception Specification

- In Java you have to inform of the exceptions that might be thrown from your method.
  - Java provides syntax (and *forces* you to use that syntax) for that
  - This is the *exception specification* and it's part of the method declaration. So your method definition might look like this:

```
void f() throws tooBig, tooSmall {
  // Some code
}
```

- exceptions of type **RuntimeException** can reasonably be thrown anywhere
- Java guarantees that exception correctness can be ensured *at compile time*.

Throwing an Exception

- Throw a different class of exception for each different type of error.
- When you throw an exception, several things happen:
  - The exception object is created in the same way that any Java object is created: on the heap, with **new**.
  - The current path of execution is stopped and the handle for the exception object is ejected from the current context.
  - The exception-handling mechanism takes over and begins to look for an appropriate place to continue executing the program.
  - This appropriate place is the *exception handler*.
  - The *exception handler* recovers from the problem so the program can either try another task or simply continue.

Throwing an Exception

- Example of throwing an exception:
  - Consider an object handle called t.
  - You might want to check before trying to call a method using that object handle if it is valid.
  - You can send information about the error into a larger context by creating an object representing your information and "throwing" it out of your current context.
  - This is called *throwing an exception*. Here's what it looks like:

```
if(t == null)
  throw new NullPointerException();
```

Catching an Exception

- If a method throws an exception, it must assume that exception is caught and dealt with.
- The try block
  - If inside a method an exception is thrown that method will exit in the process of throwing.
  - To avoid this you can set up a special block within that method to capture the exception.
  - This is called the *try block* because you "try" your various method calls there. Example:

```
try {
  // Code that might generate exceptions
}
```

Catching an Exception

- Thrown exceptions go to an *exception handler*
- Multiple *exception handlers* - one for every exception type you want to catch
- Exception handlers immediately follow the try block and are denoted by the keyword **catch**:

```
try {
    // Code that might generate exceptions
} catch(Type1 id1) {
    // Handle exceptions of Type1
} catch(Type2 id2) {
    // Handle exceptions of Type2
}
// etc...
```

Catching any Exception

- It is possible to create a handler that catches any type of exception.
- You do this by catching the base-class exception type **Exception**:

```
catch(Exception e) {
  System.out.println("caught an exception");
}
```

- The Exception class has the following methods:
  - String getMessage( )
  - String toString( )
  - void printStackTrace( )
    void printStackTrace(PrintStream)

Performing Cleanup with Finally

- If you want to execute some code whether or not an exception occurs in a try block you use a finally clause at the end of all the exception handlers:

```
try {
 // Dangerous stuff that might throw A or B
} catch (A a1) {
  // Handle A
} catch (B b1) {
  // Handle B
} finally {
  // Stuff that happens every time
}
```

- Whether an exception is thrown or not, the finally clause is always executed.

What's finally for?

- **finally** is necessary when you need to set something *other* than memory back to its original state.
- This is usually something like an open file or network connection, something you've drawn on the screen etc.
- Even in cases in which the exception is not caught in the current set of **catch** clauses, **finally** will be executed before the exception-handling mechanism continues its search for a handler at the next higher level.
- The **finally** statement will also be executed in situations in which **break** and **continue** statements are involved.

Failing to Catch an Exception

- Possible only for **RuntimeExceptions.**
- If a RuntimeException gets all the way out to **main( )** without being caught, printStackTrace( ) is called for that exception as the program exits.
- Keep in mind that it's possible to ignore only **RuntimeExceptions**, since all other handling is carefully enforced by the compiler.
- A **RuntimeException** represents a programming error:
  - An error you cannot catch (e.g. receiving a null handle handed to your method by a client programmer)
  - An error that you should have checked for in your code (such as ArrayIndexOutOfBoundsException where you should have paid attention to the size of the array).

Re-throwing an Exception

- You can re-throw the exception that you just caught:

```
catch(Exception e) {  System.out.println("An
exception was thrown");
  throw e;
}
```

  - Re-throwing an exception causes the exception to go to the exception handlers in the next-higher context.
  - Any further catch clauses for the same try block are still ignored.
  - Possible to re-throw a different exception from the one you caught.

Creating Your Own Exceptions

- Need to create your own exceptions to denote a special error that your library is capable of creating.
- To create your own exception class, you're forced to inherit from an existing type of exception.
- Inheriting an exception is quite simple:

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

Exception Restrictions

- When you override a method, you can throw only the exceptions that have been specified in the base-class version of the method.
- This is a useful restriction, since it means that code that works with the base class will automatically work with any object derived from the base class (a fundamental OOP concept, of course), including exceptions.