ENSF 593/594

11 – Exception Handling in Java

Exception Handling

- Java provides a structured means to handle errors (exceptions).
  - Less cluttered than the if-else blocks typically used in C.
- Technique:
  - If an error condition is encountered, an exception object is *thrown*.

Java Library Exceptions

- In Java there are many predefined library exception classes that we can use to handle many different anomalies. For example:
  - ArithmeticException
  - ArrayIndexOutOfBounds
  - IOException
  - …
  - And many many more

Three kind of exceptions

- **Checked Exceptions:**
  - Subject to catch or specify requirement
    - Example: IOException
- **Error:**
  - Mostly are exceptional conditions that are external to the application.
  - Cannot be anticipated or recovered
    - Example: unable to read from file due to hardware problem. Unsuccessful read throws java.io.IOError
- **Runtime exceptions:**
  - These are exceptional conditions that are internal to the application.
  - Generally, cannot be anticipated, and recovered
  - Normally logical errors due to improper use of an API:
    - Example: NullPointerException, ArithmathicException

Example of Unhandled Exception

```
public class TestJavaException
{
      // Here is an example of unhandled exception
      public static void foo ()
      {
         double x = 5, y = 0;
          double z = x /y;          // An unchecked exception
      }
      public static void main(String [] a)
      {
         foo();
      }

}
```

Catch or Specify

- There are two ways to deal with the exceptions in a method
  - To honor the catch
  - To specify the requirements

```
public void myMethod()
{
    try{
        ...
    }
    catch( AnException e)
    {
      ...
     }
}
```

```
public void myMethod() throws AnException
{

  // an operation that declares throwing AnException

}
```

Exception Handling

- An exception is *caught* by one of the following (each is tried in order):
    - A surrounding block of code
    - Some calling code
    - The JVM
- You should create an exception class for each type of error, usually by **extending** the Exception class.

Example of Handled Exception

```
public class TestJavaException {
       public static void foo (){
         try{
          double x = 5, y = 0;
           double z = x /y;          // Anomaly
        }
         catch(Arithmetic e) {
           System.out.println(“Caught an arithmetic error “ + e.getMessage());
         }

       }
       …
       …

       public static void main(String [] a)
       {
          foo();
       }
}
```

Example of Handled Exception

```
public static void foo() {
        try{
            int a, b;
            System.out.print("Enter two positive integers: ");
            Scanner scan = new Scanner(System.in);
            a = scan.nextInt();
            b = scan.nextInt();
                int z = a / b;
        }
        catch (ArithmeticException e){
            foo();
         }
 }

 public static void main(String[] args) {
        foo();
}
```

Exception Handling Advantages

1. Separating Error-Handling Code from "Regular" Code.
2. Propagating Errors Up the Call Stack
3. Grouping and Differentiating Error Types

– Example:

– Assume a Java class having a set operations on the files that should be handled.

```
open the file;
determine its size;
allocate that much memory;
read the file into memory;
close the file;
```

Pseudocode – A Traditional Style Error Handling

```
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;
```

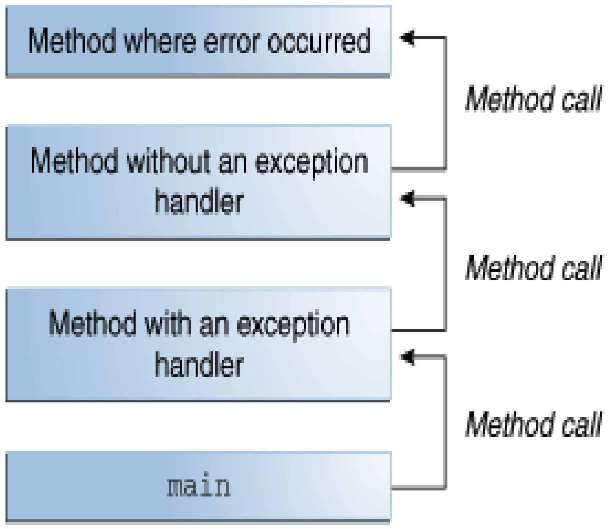
Java Style Exception Handling

```
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;
  }
```

Psuedocode - Propagating the Error Up the Call Stack

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}

errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}

errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```


Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main

Grouping and Differentiating Error Type

- **Handle a specific handler**

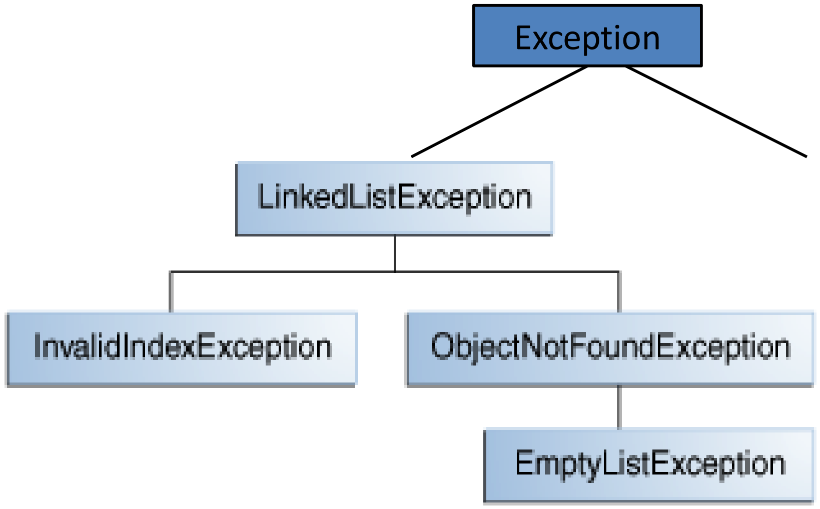
```
catch (EmptyListException e) {
    ...
}
```

- **Handle all in a hierarchy**

```
catch (IOException e) {
    ...
}
```

- **Handle any exception**

```
catch (Excetion e) {
    ...
}
```



Defining User-Defined Exceptions

Implementation of a User-defined Exception

```
class SizeableException extends Exception {
    public SizeableException(){

          super("Size limit exceeded.");
    }
  }
```

constructor

creates a message

Throwing an Exception

```
class Circle {
  private radius;
  ...
  public void enlarge(int s) throws SizeableException
    {
        if ( s < 0 || radius < 0)
           throw new SizeableException();

         radius *= s;
    }

}
```

the method will throw this exception if it encounters an error

a new exception object is instantiated and thrown, if the moon to be consumed is radioactive.

Handling Exceptions

- You can have zero or more catch clauses.
  - Each must catch a different exception.
- The finally clause is optional.
  - It is *always* executed after code in the try or catch blocks is executed
    - What if there is return in the try block
    - What if there there is a System.exit(1) in the try block.
- In a hierarchy of exception classes the order of catch clauses should go from specific to more general.

Handling Exceptions - Example

```
static public void main (String [] args)  {

 Circle c = new Circle(60);
 try {
         c.enlarge(2);
         ...    // MORE CODE TO
   }
   catch(SizeableException e) {
         System.out.println("Error: ... " +  e.getMessage());
   }
   catch(OtherExceptions oe){
     ...
   }
   catch(Exception oe){
     ...
   }
   finally {
     ...
   }
  …
}
```

Tracing Exception Information

```
catch (Exception e) {
    StackTraceElement elements[] = e.getStackTrace();
    for (int i = 0, n = elements.length; i < n; i++) {
        System.err.println(elements[i].getFileName()
          + ":" + elements[i].getLineNumber()
          + ">> ”  + elements[i].getMethodName() + "()");
    }
}
```