Class 14 Notes

Review

An ***Exception*** is an abnormal condition that occurs at runtime.

Example:

- Null pointer Exception
- Array out of bounds Exception
- Type mismatch exception

Java provides a mechanism to handle exceptions explicitly:

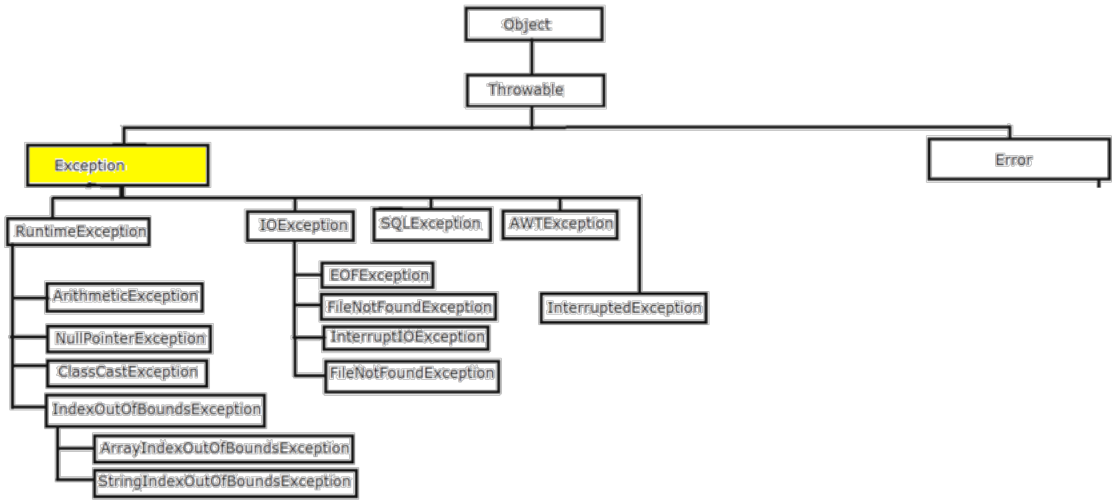
the try-catch-throw construction

Here is what happens when an exception occurs:

1. An Exception object is created. This is a genuine object and like all objects this holds information about the exception. The Exception object is created by the Java Virtual machine or the program.

2. The Exception object is passed ("thrown") to a "catch block." The catch block handles the exception. The catch block is a section of code.

3. The program continues with the code following the catch block.
4. Java has a hierarchy of Exception classes. Here is a partial picture of the Exception hierarchy:



Note:
A RuntimeException is-a Exception
An ArithmeticException is-a Exception

All of the classes under Exception ac be upcast to Exception

An Exception object can be thrown

1. **Explicitly by your program** as in the Bank Account program

2. **By the JVM.** If the JVM throws the Exception there is no explicit "throw statement."

Here is an example where the program explicitly throws and catches an Exception

```
import java.util.*;
public class ProgramHandlesExceptions
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        boolean correct = false;
        while (!correct)
        {
          correct = true;
          String numberString ="";
          try
          {
             System.out.print("Enter a number ");
             numberString = input.next();
             for (int i = 0; i < numberString.length(); i++)
                if (!Character.isDigit(numberString.charAt(i)))
                {
                   Exception e = new Exception("Evil input "+ numberString + " reenter ");
                    throw e;
                }
          }
            catch (Exception e)
            {
              System.out.println(e.getMessage());
              correct = false;
            }
            if (correct)
            System.out.println(" You entered "+ numberString);
          } // end while
     }
}
```

Here is an example where an exception thrown by the Java Virtual Machine is caught in the program. Notice no *throw* statement:

```
import java.io.*;
public class CatchMeAgain
{
   //JVM throws the exception, program catches it
   // Notice no "throws IOException"

 public static void main(String[] args)
 {
  File f = new File("realfile.txt");
  int sum = 0;

  try
  {
    Scanner in = new Scanner(f);  // possible exception here
    while( in.hasNext())
            sum = sum+ in.nextInt()
  }
  catch (Exception e)
  {
     System.out.println("Doh! Bad file: "+
                      f.getName()+ "   "+ e.getMessage());
     System.exit(0);
  }
  System.out.println("The sum is "+ sum);
 }
}
```

How throw-catch-try works

- When an Exception occurs in a try block, program control immediately passes to the catch block.
- The code in the catch block is executed and
- The program continues with the code following the catch block.
- Once an Exception is thrown in a try block, the remaining code in the try block is skipped.

Multiple Catch Blocks

Several catch blocks can be associated with a single try block. For example,

```
try
{
        statements
}
catch ( ArithmeticException e)
{
        statements
}
catch ( NullPointerException e)
{
        statements
}
catch ( Exception e)
{
        statements
}
```

In this case, the **first catch block** with parameter matching the type of thrown exception catches the exception.

```
public class CatchMe
{
   // Java throws the exception
   // Notice no "throws IOException"

 public static void main(String[] args)
 {
  int n = 0;
  boolean correct = false;
  String filename = "";
  File f= null;
  Scanner input = new Scanner(System.in);
```

```
while (!correct)
  {
   try
   {
     System.out.print("Enter a filename:");
     filename = input.nextLine();
     f = new File(filename);
     Scanner fileInput = new Scanner(f);
                                   // possible  FileNotFoundException
      correct = true;
     n = fileInput.nextInt();  // possible InputMismatchException
   }

   catch (FileNotFoundException e)
   {
    System.out.println("Doh! Bad file: "+ f.getName());
   }
   catch (InputMismatchException e)
   {
    System.out.println("Doh! Bad data in: "+ f.getName());
   }
 }
  System.out.println("The square is "+ (n*n));
 }
}
```

The following fragment prints the square root of a (non-negative) number. Exceptions occur when the user enters a negative number or, possibly, a non-numeric string.

```
 try
{
        System.out.print("Enter an integer: ");
        String number = input.next();
        int value = Integer.parseInt(number);      // possible NumberFormatException

         if (y < 0)
                throw new Exception(" Input Error: Negative Number");   // Exception if y is negative
          else
                System.out.println("Square root: " + (Math.sqrt(y)));
 }

 catch (NumberFormatException e)
{
        System.out.println("Illegal number format ");
}

catch (Exception e)
{
        System.out.println( e.getMessage());
}
```

If a user enters "*abcd* " as input, the Java Virtual Machine throws a NumberFormatException when parseInt(...) is called.

The first catch block catches and handles this exception.

On the other hand, if the user input is –54, the statement

```
if (y < 0)
  throw new Exception("Negative Number.  Reenter");
```

throws an Exception object.

- The first catch block does not catch this exception since the parameter of the first catch block is of type **NumberFormatException**.
- The second catch block with parameter type **Exception** does, in fact, catch the exception.
- Actually , the final catch block catches ***any*** exception that is not caught by preceding catch blocks. The final catch block is a "catch all."
- The catch blocks are purposely written in order from most specific to least specific. If the "catch(Exception e) block" had come first, then all exceptions would be caught by that block, and the code would not distinguish a NumberFormatException from another type of Exception. Exception is the superclass of all other exceptions.

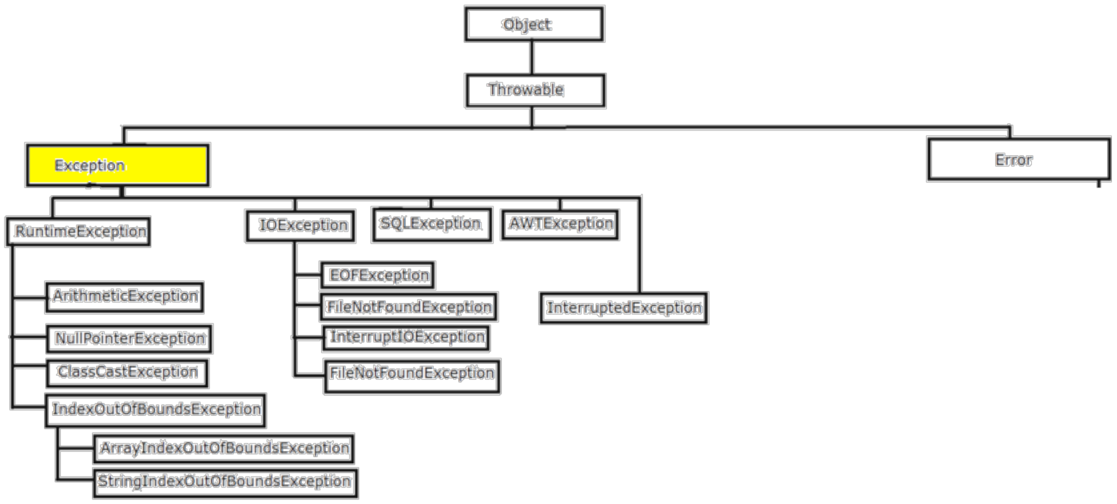
Checked and Unchecked Exceptions

Java divides its Exception hierarchy into two groups:

- Checked exceptions
- Unchecked Exceptions

Runtime Exceptions fall into the Unchecked Exceptions group. These include
ArithmeticException
ArrayIndexOutOfBoundsException
NullPointerException
StringIndexOutOfBoundsException

Here is a picture any class under RnntimeException is an unchecked exception. Others are checked exceptions



Catching an unchecked Exception is optional.

All other exceptions are checked. Your program is required to handle checked exceptions. For, our programs, these are usually just IOExceptions.

If a method does not explicitly catch and handle a checked exception, the method, by including a throws clause in its heading passes the exception back to the caller and it becomes the caller's responsibly to handle the exception.

Checked exceptions can be passed along the chain of method calls right up to the main() method and finally to the system until they are eventually caught.

Here *readData(…)* throws and also catches a possible exception

```
import java.util.*;
import java.io.*;
public class File1
{
     public static void readData(String fileName)
    {
        try
        {
            File inputFile = new File(fileName);
            Scanner input = new Scanner(inputFile);  // possible exception
            String line;
            while (input.hasNext())
            {
                line =  input.nextLine();
                System.out.println(line);
            }
            input.close();
        }
        catch (FileNotFoundException e)
        {
            System.out.println("Error: File not found: " + fileName);
        }
    }

    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        System.out.print("Input file: ");
        String fileName = input.next();
        readData(fileName);
    }
}
```

Notice: The Exception was handled – no throws clause

Here *readData(…)* does not catch the Exception but throws it to the caller, *main(…).* ReadData has a throws clause

- **The *main(…)* method catches the exception**
- **Notice that main has a try and a catch block.**

```
import java.util.*;
import java.io.*;
public class File2
{
    public static void readData(String fileName) throws FileNotFoundException
    {
        File inputFile = new File(fileName);
        Scanner input = new Scanner(inputFile);
        String line;
        while (input.hasNext())
        {
            line =  input.nextLine();
            System.out.println(line);
        }
        input.close();
    }
    public static void main(String[] args)  // look no throws, handles the exception
    {
        Scanner input = new Scanner(System.in);
        System.out.print("Input file: ");
        String fileName = input.next();
        try
        {
            readData(fileName);   // can throw an exception
        }
        catch (FileNotFoundException e)
        {
            System.out.println("File not found : " + fileName);
            System.out.println("Program terminated");
        }
    }
}
```

- **Here *readData(…)* throws the Exception back to *main(…)***
- ***main(…)* does not handle the exception and throws it to the system**

```
import java.util.*;
import java.io.*;
public class File3
{
    public static void readData(String fileName) throws FileNotFoundException
                                                                                   // to caller
   {
      File inputFile = new File(fileName);
      Scanner input = new Scanner(inputFile);
      String line;
      while (input.hasNext())
      {
         line =  input.nextLine();
         System.out.println(line);
      }
      input.close();
   }
   public static void main(String[] args) throws FileNotFoundException   // to system
   {
      Scanner input = new Scanner(System.in);
      System.out.print("Input file: ");
      String fileName = input.next();
      readData(fileName);
   }
}
```

Here are some of the possibilities
Here main() calls A() and A() calls B() that is main(..)→A()→ B()
Assume an IOException can occur in method B()

- **B() throws the IOException back to A()**
- **A() throws the IOException back to main(..)**
- **main(…) throws it back to the System**
- **The system handles the IOException**

```
public class Dumb
{
  public void B() throws IOException
  {
      // throws back to A()
      // code
  }
   public void A() throws IOException
  {
     // throws back to MAIN()

      B();
      // code
  }

  public static void main(…) throws IOException
  {
      A();
      // code
  }
 }
```

- **B() throws the IOException back to A()**
- **A() handles the IOException**
  main() does not have a throws

```
public class Dumb
{
  public void B() throws IOException
  {
      // throws back to A()
      // code
  }

   public void A()  // handles IOException
  {

      try
     {
         B();
     }
     catch (IOException e)
      { ///catch block code
      }
  }

  public static void main(…)
  {
      A();
      // code
  }}
```

- **B() throws the IOException back to A()**
- **A() throws the IOException back to main(..)**
- **main() handles the IOException**

```
public class Dumb
{
  public void B() throws IOException
  {
      // throws back to A()
      // code
  }
   public void A() throws IOException
  {
     // throws back to main()
      B();
      // code
  }
```

```
public static void main(…)

  {
     try
     {
      A();
      // code
     }
     catch (IOException e)
      { ///catch block code
      }

  }
}
```

Extending Exception: Writing your own exception class:

```
import java.util.*;
class NumberOutOfRangeException extends Exception
{
        public NumberOutOfRangeException()
        {
           super("Number out of Range");  // call one arg constructor of Exception
        }
        public NumberOutOfRangeException(String s)
        {
                    super(s);     // calls the one arg constructor of Exception
        }
}

public class DumbException
{
  public static void main(String[] args)
  {
        Scanner input = new Scanner(System.in);
        Random random = new Random();
        int number = random.nextInt(100)+ 1;;
        int guess = 0;
        int numGuesses = 0;
        while (guess != number)
        {
                try
                {
                        System.out.print("Guess a number between 1 and 100: ");
                        guess = input.nextInt();
                        if (guess < 1 || guess > 100)
                                throw( new NumberOutOfRangeException());
                         numGuesses++;
                          if (guess > number)
                                System.out.println("Too high");
                        else if (guess < number)
                                System.out.println("Too low");
                        else
                         {
                                System.out.println("That's it");
                                System.out.println("Number of tries: "+ numGuesses);
                        }
                }
                catch( NumberOutOfRangeException e)
                {
                        System.out.println( guess + " "+ e.getMessage());
                }
        }
    }
}
```

The finally block:

The ***finally block*** is a block of code that always executes, regardless of whether or not an exception is thrown.

A finally block is always paired with a try-catch or just a try block. A finally block is usually used to release system resources such as closing a Scanner or PrintWriter.

```
import java.util.*;
import java.io.*;
public class FinallyBlock
{
public static void fileStuff(String name1, String name2)
{
        Scanner input= null;
        PrintWriter output= null;
        try
        {
                File file1 = new File(name1); //input file
                File file2 = new File(name2); // output file
                input = new Scanner (file1);
                output = new PrintWriter(file3);
        // code to perform some task with files
        }
        catch (IOException e)
        {
                System.out.println("Error in method fileStuff());
        }
        finally
        {
                if ( input != null)
                        input1.close();
                if (output != null)
                        output.close();
                System.out.println("Finally block completed ");
        }
}
public static void main (String[] args)
{
        Scanner input = new Scanner(System.in);
        String name1, name2, name3;
        System.out.print("File 1: ");
        name1 = input.next();
        System.out.print("File 2: ");
        name2 = input.next();
        fileStuff( name1,name2);
}
```

```
import java.util.*;
import java.io.*;
public class YesFinally
{
  public static void main(String[] args)
  {
    Scanner input = new Scanner(System.in);
    Scanner infile = null;;
    PrintWriter pw = null;
    System.out.print("Enter file name: ");
     String filename = input.nextLine();
     try
     {
       File in = new File(filename);
        infile = new Scanner(in);
       File outfile = new File("outfile.txt");
       pw = new PrintWriter(outfile);
       while ( infile.hasNext())
       {
         int x = infile.nextInt();
         pw.println(x);
       }
     }

     catch (FileNotFoundException e)
     {
        System.out.println("File not Found");
     }
     catch( Exception e)
     {
       System.out.println("Incorrect data");
    }

     finally
     {
       infile.close();
       pw.close();
     }
 }
}
```

Here the finally block closes the scanner and printwriter. If you do not close the printwriter some data in the output may be lost

What is the output? Notice there is no catch.block. You can pair finally with a try block.

```
public class Mystery
{
 public boolean mystery()
 {
  try
  {
   return true;
  }
  finally
  {
   return false;
  }
 }

 public static void main(String[] args)
 {
  Mystery m = new Mystery();
  System.out.println(m.mystery());
 }
}
```