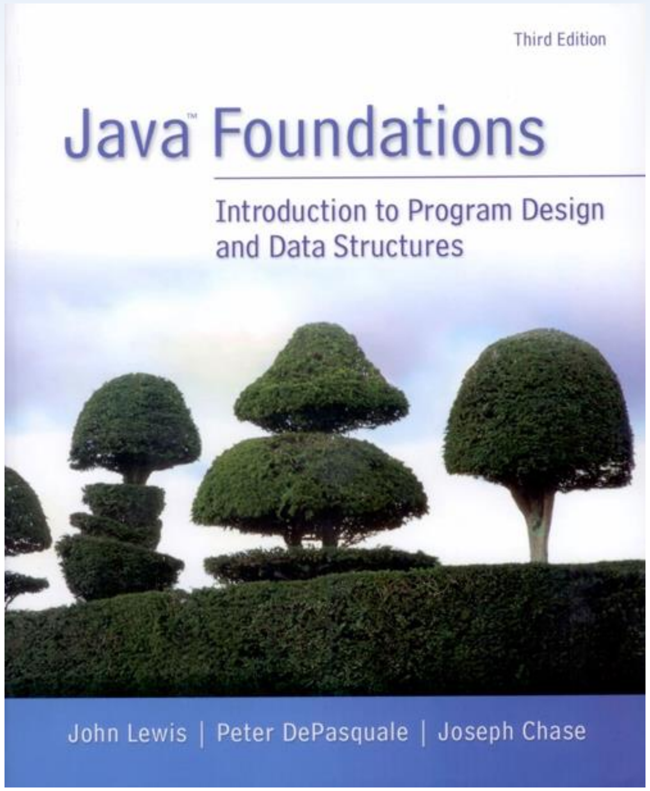

Third Edition
Java™ Foundations
Introduction to Program Design and Data Structures
John Lewis | Peter DePasquale | Joseph Chase

Chapter 10

Exceptions

Chapter Scope

- The purpose of exceptions
- Exception messages
- The call stack trace
- The try-catch statement
- Exception propagation
- The exception class hierarchy
- I/O exceptions (and writing text files)

Exceptions

- An *exception* is an object that describes an unusual or erroneous situation
- Exceptions are *thrown* by a program, and may be *caught* and *handled* by another part of the program
- A program can be separated into a normal execution flow and an *exception execution flow*
- An *error* is also represented as an object in Java, but usually represents a unrecoverable situation and should not be caught

Exception Handling

- The Java API has a predefined set of exceptions and errors that can occur during execution
- A program can deal with an exception in one of three ways:
  - ignore it
  - handle it where it occurs
  - handle it an another place in the program
- The manner in which an exception is processed is an important design consideration

Uncaught Exceptions

- If an exception is ignored by the program, the program will terminate abnormally and produce an appropriate message
- The message includes a *call stack trace* that
  - indicates the line on which the exception occurred
  - shows the method call trail that lead to the attempted execution of the offending line

```
//********************************************************************
//  Zero.java       Java Foundations
//
//  Demonstrates an uncaught exception.
//********************************************************************

public class Zero
{
   //-----------------------------------------------------------------
   //  Deliberately divides by zero to produce an exception.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      int numerator = 10;
      int denominator = 0;

      System.out.println("Before the attempt to divide by zero.");

      System.out.println(numerator / denominator);

      System.out.println("This text will not be printed.");
   }
}
```

The try-catch Statement

- To handle an exception in a program, the line that throws the exception is executed within a *try block*
- A try block is followed by one or more *catch* clauses
- Each catch clause has an associated exception type and is called an *exception handler*
- When an exception occurs, processing continues at the first catch clause that matches the exception type

```java
//********************************************************************
//  ProductCodes.java       Java Foundations
//
//  Demonstrates the use of a try-catch block.
//********************************************************************

import java.util.Scanner;

public class ProductCodes
{
   //-----------------------------------------------------------------
   //  Counts the number of product codes that are entered with a
   //  zone of R and and district greater than 2000.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      String code;
      char zone;
      int district, valid = 0, banned = 0;

      Scanner scan = new Scanner(System.in);

      System.out.print("Enter product code (STOP to quit): ");
      code = scan.nextLine();
      while (!code.equals("STOP"))
      {
         try
         {
            zone = code.charAt(9);
            district = Integer.parseInt(code.substring(3, 7));
            valid++;
            if (zone == 'R' && district > 2000)
               banned++;
         }
         catch (StringIndexOutOfBoundsException exception)
         {
            System.out.println("Improper code length: " + code);
         }
         catch (NumberFormatException exception)
         {
            System.out.println("District is not numeric: " + code);
         }

         System.out.print("Enter product code (STOP to quit): ");
         code = scan.nextLine();
      }

      System.out.println("# of valid codes entered: " + valid);
      System.out.println("# of banned codes entered: " + banned);
   }
}
```

The finally Clause

- A try statement can have an optional clause following the catch clauses, designated by the reserved word `finally`
- The statements in the finally clause always are executed
- If no exception is generated, the statements in the finally clause are executed after the statements in the try block complete
- If an exception is generated, the statements in the finally clause are executed after the statements in the appropriate catch clause complete

Exception Propagation

- An exception can be handled at a higher level if it is not appropriate to handle it where it occurs
- Exceptions *propagate* up through the method calling hierarchy until they are caught and handled or until they reach the level of the `main` method
- A try block that contains a call to a method in which an exception is thrown can be used to catch that exception

```java
//********************************************************************
//  Propagation.java       Java Foundations
//
//  Demonstrates exception propagation.
//********************************************************************

public class Propagation
{
   //-----------------------------------------------------------------
   //  Invokes the level1 method to begin the exception demonstration.
   //-----------------------------------------------------------------
   static public void main(String[] args)
   {
      ExceptionScope demo = new ExceptionScope();

      System.out.println("Program beginning.");
      demo.level1();
      System.out.println("Program ending.");
   }
}
```

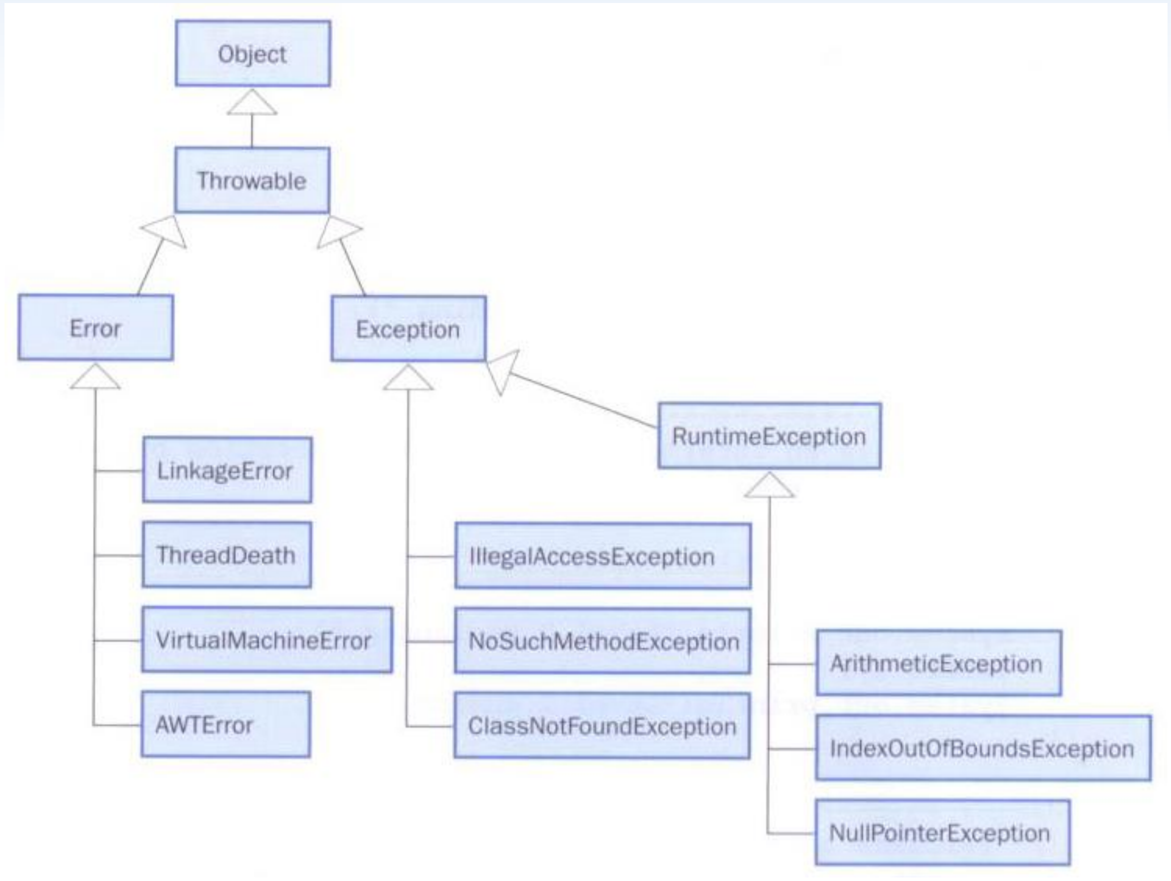
```
//********************************************************************
//  ExceptionScope.java       Java Foundations
//
//  Demonstrates exception propagation.
//********************************************************************

public class ExceptionScope
{
   //-----------------------------------------------------------------
   //  Catches and handles the exception that is thrown in level3.
   //-----------------------------------------------------------------
   public void level1()
   {
      System.out.println("Level 1 beginning.");

      try
      {
         level2();
      }
      catch (ArithmeticException problem)
      {
         System.out.println();
         System.out.println("The exception message is: " +
                            problem.getMessage());
         System.out.println();
         System.out.println("The call stack trace:");
         problem.printStackTrace();
         System.out.println();
      }

      System.out.println("Level 1 ending.");
   }
   //-----------------------------------------------------------------
   //  Serves as an intermediate level.  The exception propagates
   //  through this method back to level1.
   //-----------------------------------------------------------------
   public void level2()
   {
      System.out.println("Level 2 beginning.");
      level3();
      System.out.println("Level 2 ending.");
   }

   //-----------------------------------------------------------------
   //  Performs a calculation to produce an exception.  It is not
   //  caught and handled at this level.
   //-----------------------------------------------------------------
   public void level3()
   {
      int numerator = 10, denominator = 0;

      System.out.println("Level 3 beginning.");
      int result = numerator / denominator;
      System.out.println("Level 3 ending.");
   }
}
```

The Exception Class Hierarchy

- Classes that define exceptions are related by inheritance, forming an exception class hierarchy
- All error and exception classes are descendents of the `Throwable` class
- A programmer can define an exception by extending the `Exception` class or one of its descendants
- The parent class used depends on how the new exception will be used

The Exception Class Hierarchy

- Part of the error and exception class hierarchy in the Java API:



Checked Exceptions

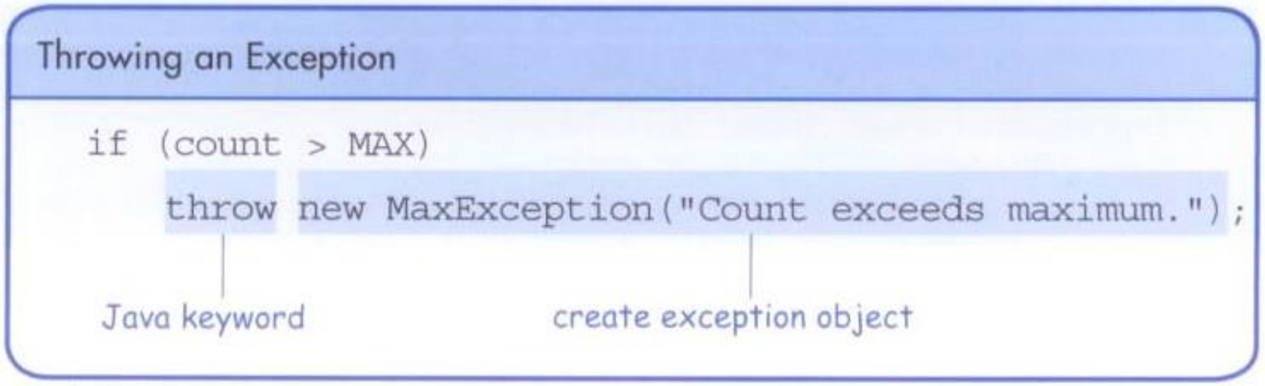
- An exception is either *checked* or *unchecked*
- A *checked exception* either must be caught by a method, or must be listed in the *throws clause* of any method that may throw or propagate it
- A throws clause is appended to the method header
- The compiler will issue an error if a checked exception is not caught or asserted in a throws clause

Unchecked Exceptions

- An unchecked exception does not require explicit handling, though it could be processed that way
- The only unchecked exceptions in Java are objects of type `RuntimeException` or any of its descendants
- Errors are similar to `RuntimeException` and its descendants in that:
  - Errors should not be caught
  - Errors do not require a throws clause

The throw Statement

- Exceptions are thrown using the *throw* statement
- Usually a throw statement is executed inside an if statement that evaluates a condition to see if the exception should be thrown


Throwing an Exception

```
if (count > MAX)
    throw new MaxException("Count exceeds maximum.");
```

```java
//********************************************************************
//  CreatingExceptions.java       Java Foundations
//
//  Demonstrates the ability to define an exception via inheritance.
//********************************************************************

import java.util.Scanner;

public class CreatingExceptions
{
   //-----------------------------------------------------------------
   //  Creates an exception object and possibly throws it.
   //-----------------------------------------------------------------
   public static void main(String[] args) throws OutOfRangeException
   {
      final int MIN = 25, MAX = 40;

      Scanner scan = new Scanner(System.in);

      OutOfRangeException problem =
         new OutOfRangeException("Input value is out of range.");

      System.out.print("Enter an integer value between " + MIN +
                       " and " + MAX + ", inclusive: ");
      int value = scan.nextInt();

      //  Determine if the exception should be thrown
      if (value < MIN || value > MAX)
         throw problem;

      System.out.println("End of main method.");  // may never reach
   }
}
```

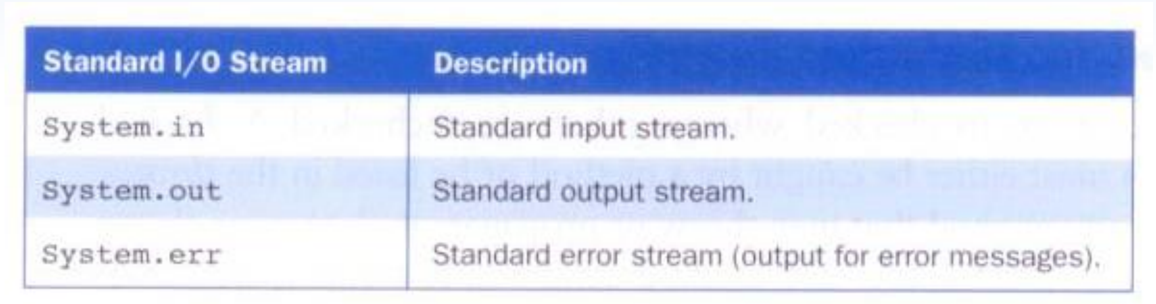
```
//********************************************************************
//  OutOfRangeException.java       Java Foundations
//
//  Represents an exceptional condition in which a value is out of
//  some particular range.
//********************************************************************

public class OutOfRangeException extends Exception
{
   //-----------------------------------------------------------------
   //  Sets up the exception object with a particular message.
   //-----------------------------------------------------------------
   OutOfRangeException(String message)
   {
      super(message);
   }
}
```

I/O Exceptions

- Let's examine issues related to exceptions and I/O
- A *stream* is a sequence of bytes that flow from a source to a destination
- In a program, we read information from an input stream and write information to an output stream
- A program can manage multiple streams simultaneously

Standard I/O

- There are three standard I/O streams:



Standard I/O Stream	Description
`System.in`	Standard input stream.
`System.out`	Standard output stream.
`System.err`	Standard error stream (output for error messages).

- We use `System.out` when we execute `println` statements
- `System.out` and `System.err` typically represent a particular window on the monitor screen
- `System.in` typically represents keyboard input, which we've used with `Scanner` objects

The IOException Class

- Operations performed by some I/O classes may throw an `IOException`
  - A file might not exist
  - Even if the file exists, a program may not be able to find it
  - The file might not contain the kind of data we expect
- An `IOException` is a checked exception

Writing Text Files

- In Chapter 4 we explored the use of the `Scanner` class to read input from a text file
- Let's now examine other classes that let us write data to a text file
- The `FileWriter` class represents a text output file, but with minimal support for manipulating data
- Therefore, we also rely on `PrintWriter` objects, which have `print` and `println` methods

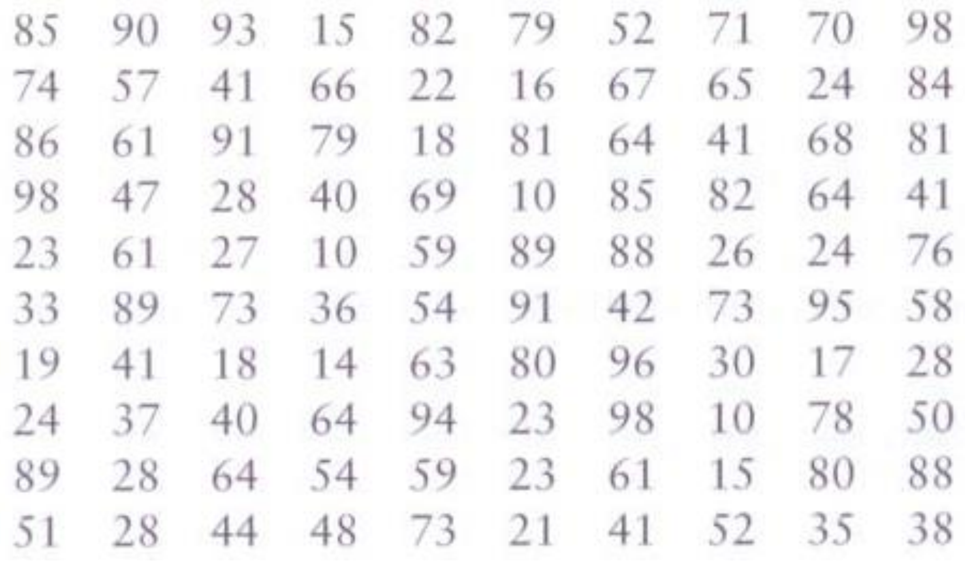
Writing Text Files

- We build the class that represents the output file by combining these classes appropriately
- Output streams should be closed explicitly
- Let's look at a program that writes a test data file with random 2-digit numbers

```
//********************************************************************
//  TestData.java       Java Foundations
//
//  Demonstrates I/O exceptions and the use of a character file
//  output stream.
//********************************************************************

import java.util.Random;
import java.io.*;

public class TestData
{
   //-----------------------------------------------------------------
   //  Creates a file of test data that consists of ten lines each
   //  containing ten integer values in the range 10 to 99.
   //-----------------------------------------------------------------
   public static void main(String[] args) throws IOException
   {
      final int MAX = 10;

      int value;
      String file = "test.dat";

      Random rand = new Random();

      FileWriter fw = new FileWriter(file);
      BufferedWriter bw = new BufferedWriter(fw);
      PrintWriter outFile = new PrintWriter(bw);
      for (int line=1; line <= MAX; line++)
      {
         for (int num=1; num <= MAX; num++)
         {
            value = rand.nextInt(90) + 10;
            outFile.print(value + "   ");
         }
         outFile.println();
      }

      outFile.close();
      System.out.println("Output file has been created: " + file);
   }
}
```

TestData Example

- Sample data written to the file:



```
85 90 93 15 82 79 52 71 70 98
74 57 41 66 22 16 67 65 24 84
86 61 91 79 18 81 64 41 68 81
98 47 28 40 69 10 85 82 64 41
23 61 27 10 59 89 88 26 24 76
33 89 73 36 54 91 42 73 95 58
19 41 18 14 63 80 96 30 17 28
24 37 40 64 94 23 98 10 78 50
89 28 64 54 59 23 61 15 80 88
51 28 44 48 73 21 41 52 35 38
```