Exception Handling

CSE 114: Introduction to Object-Oriented Programming

Paul Fodor

Stony Brook University

http://www.cs.stonybrook.edu/~cse114

Contents

- Motivation for Exception Handling: invalid inputs and external errors, so that we avoid runtime errors and the program can continue to run by handling exceptions or terminate gracefully.
- Exception Classes
  - Unchecked Exceptions vs. Checked Exceptions
- Declaring, Throwing, and Catching Exceptions
  - Catch or Declare Checked Exceptions
- The **`finally`** Clause
- When To Use Exceptions
- Defining and Using Custom Exception Classes
- Text I/O: The **`File`** Class
  - Writing Data Using **`PrintWriter`**
  - Reading Data Using **`Scanner`**

Motivation

- When a program runs into a **runtime error**, the program terminates abnormally.
  - We want to handle the runtime error so that the program can continue to run or terminate gracefully.

Why use Exception-Handling?

Runtime error: the user enters an invalid input when asked for an int

```
import java.util.Scanner;
public class TestReadNonIntAsInt {
      public static void main(String[] args) {
            Scanner input = new Scanner(System.in);
            System.out.print("Enter an int: ");
            int num = input.nextInt();
            System.out.println(num);
      }
}
```

```
Enter an int: x
Exception in thread "main"
java.util.InputMismatchException
at java.util.Scanner.throwFor(Unknown Source)
at java.util.Scanner.next(Unknown Source)
at java.util.Scanner.nextInt(Unknown Source)
at java.util.Scanner.nextInt(Unknown Source)
at TestReadNonIntAsInt.main(TestReadNonIntAsInt.java:7)
```

This looks bad to a user (e.g., a medical professional, financial analyst, etc.)

Why use Exception-Handling?

It does not matter if we do it differently:

```
import java.util.Scanner;
public class TestParseNonIntAsInt {
      public static void main(String[] args) {
            Scanner input = new Scanner(System.in);
            System.out.print("Enter an int: ");
            String s = input.next();
            int num = Integer.parseInt(s);
            System.out.println(num);
      }
}
```

```
Enter an int: x
Exception in thread "main" java.lang.NumberFormatException:
For input string: "abc"
at java.lang.NumberFormatException.forInputString(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at java.lang.Integer.parseInt(Unknown Source)
at TestParseNonIntAsInt.main(TestParseNonIntAsInt.java:7)
```

Why use Exception-Handling?

Runtime errors: division with 0 (if the user enters 1 and 0)

```
import java.util.Scanner;

public class Quotient {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();
    System.out.println(number1 + " / " + number2
      + " is " +(number1 / number2));
  }
}
```

```
Enter two integers: 1 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Quotient.main(Quotient.java:10)
```

Why use Exception-Handling?

What if the runtime error occurs in a method?

```
import java.util.Scanner;
public class QuotientWithMethod {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();
    int result = quotient(number1, number2);
    System.out.println(number1 + " / " + number2 + " is " + result);
    System.out.println("Execution continues ...");
  }
  public static int quotient(int number1, int number2) {
    return number1 / number2;
  }
}
```

```
Enter two integers: 1 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Test.quotient(Test.java:15)
at Test.main(Test.java:10)
```

Exception-Handling

Exception-Handling in the same method:

```
import java.util.Scanner;
public class QuotientWithException {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();
    try {
       System.out.println(number1 + " / " + number2 + " is "
             + (number1 / number2));
    }catch (Exception ex) {
       System.out.println("Exception: " +
         "an integer cannot be divided by zero ");
    }
    System.out.println("Execution continues ...");
  }
}
```

Exception-Handling

Declaring and throwing exceptions in a method

```
import java.util.Scanner;
public class QuotientWithMethod {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();
    try {
      int result = quotient(number1, number2);
      System.out.println(number1 + " / " + number2 + " is " + result);
    }catch (Exception ex) {
      System.out.println("Exception: an integer " +
        "cannot be divided by zero ");
    }
    System.out.println("Execution continues ...");
  }
  public static int quotient(int number1, int number2) throws Exception{
    if (number2 == 0)
      throw new ArithmeticException("Divisor cannot be zero");
    return number1 / number2;
  }
}
```

Exception Advantages

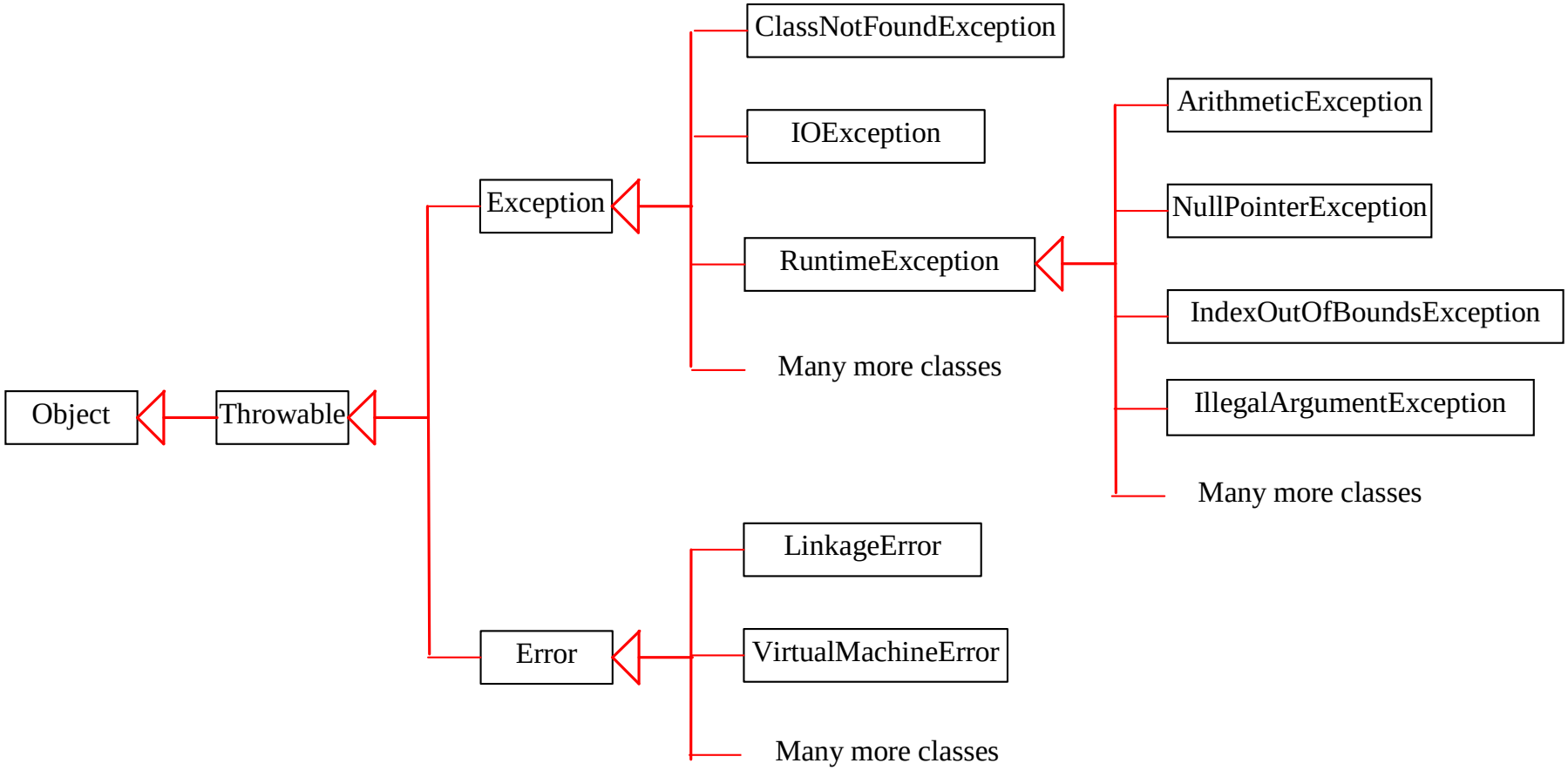
- It enables a method to throw an exception to its caller.
  - Without this capability, a method must handle the exception itself (and return an incorrect value) or terminate the program.

Handling invalid inputs

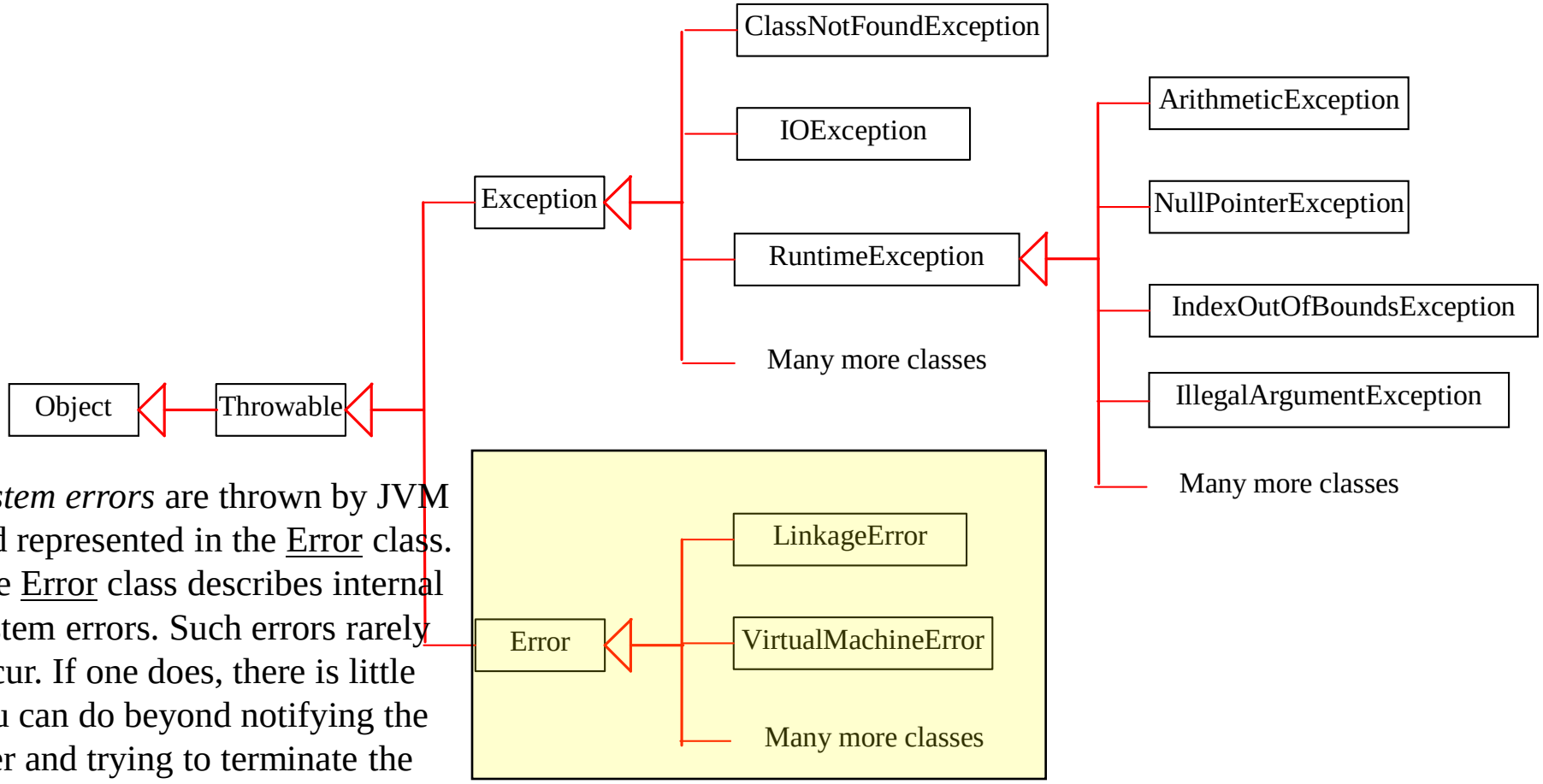
- By handling an exception, your program can continuously read an input until it is correct:

```
import java.util.*;
public class InputMismatchExceptionDemo {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    boolean continueInput = true;
    do {
      try {
        System.out.print("Enter an integer: ");
        int number = input.nextInt();
        // Display the result
        System.out.println("The number entered is " + number);
        continueInput = false;
      } catch (InputMismatchException ex) {
        System.out.println("Try again. Incorrect input: " +
            "an integer is required.");
        input.nextLine(); // discard input
      }
    } while (continueInput);
  }
}
```

Exception Classes



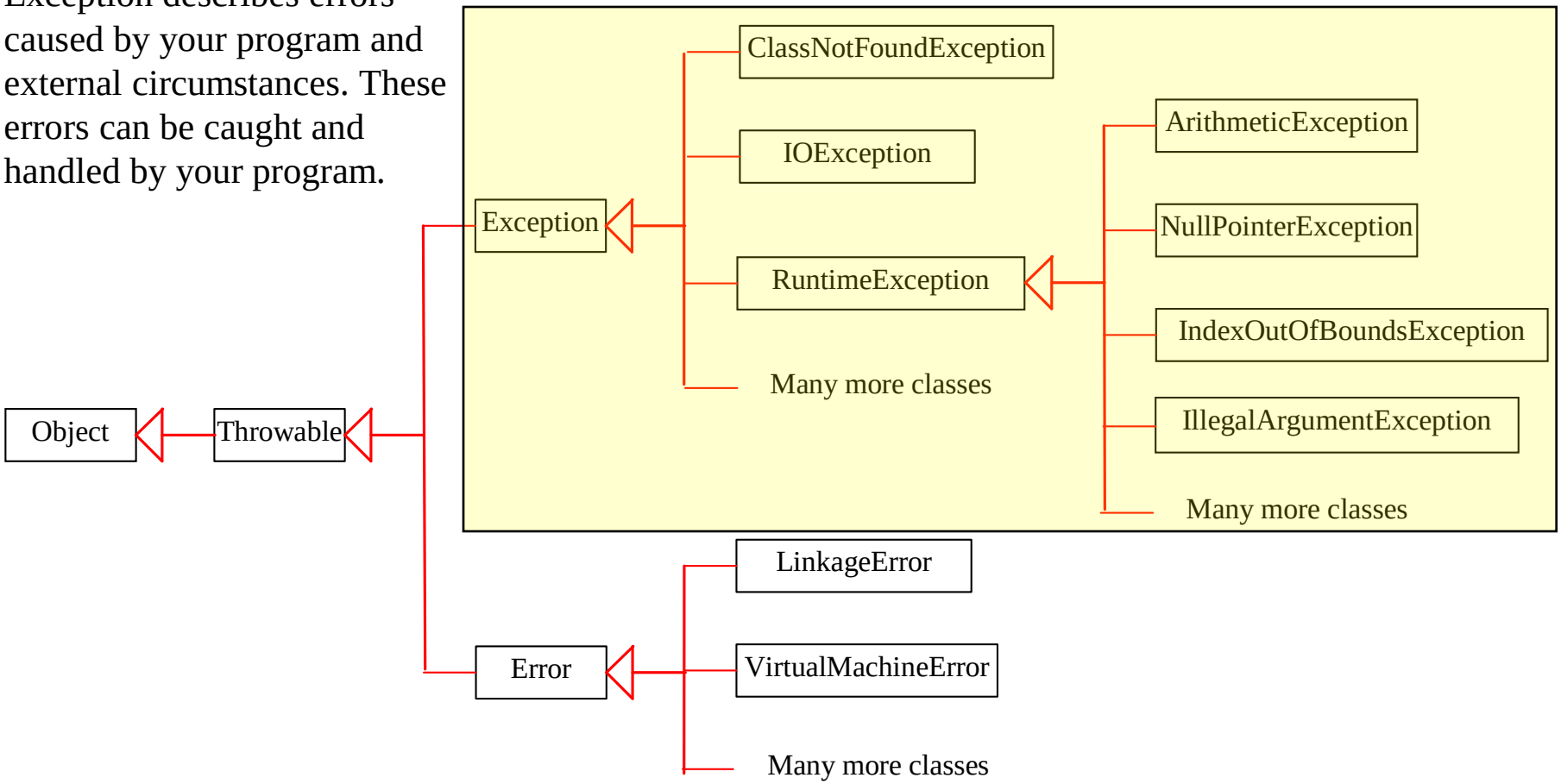
System Errors



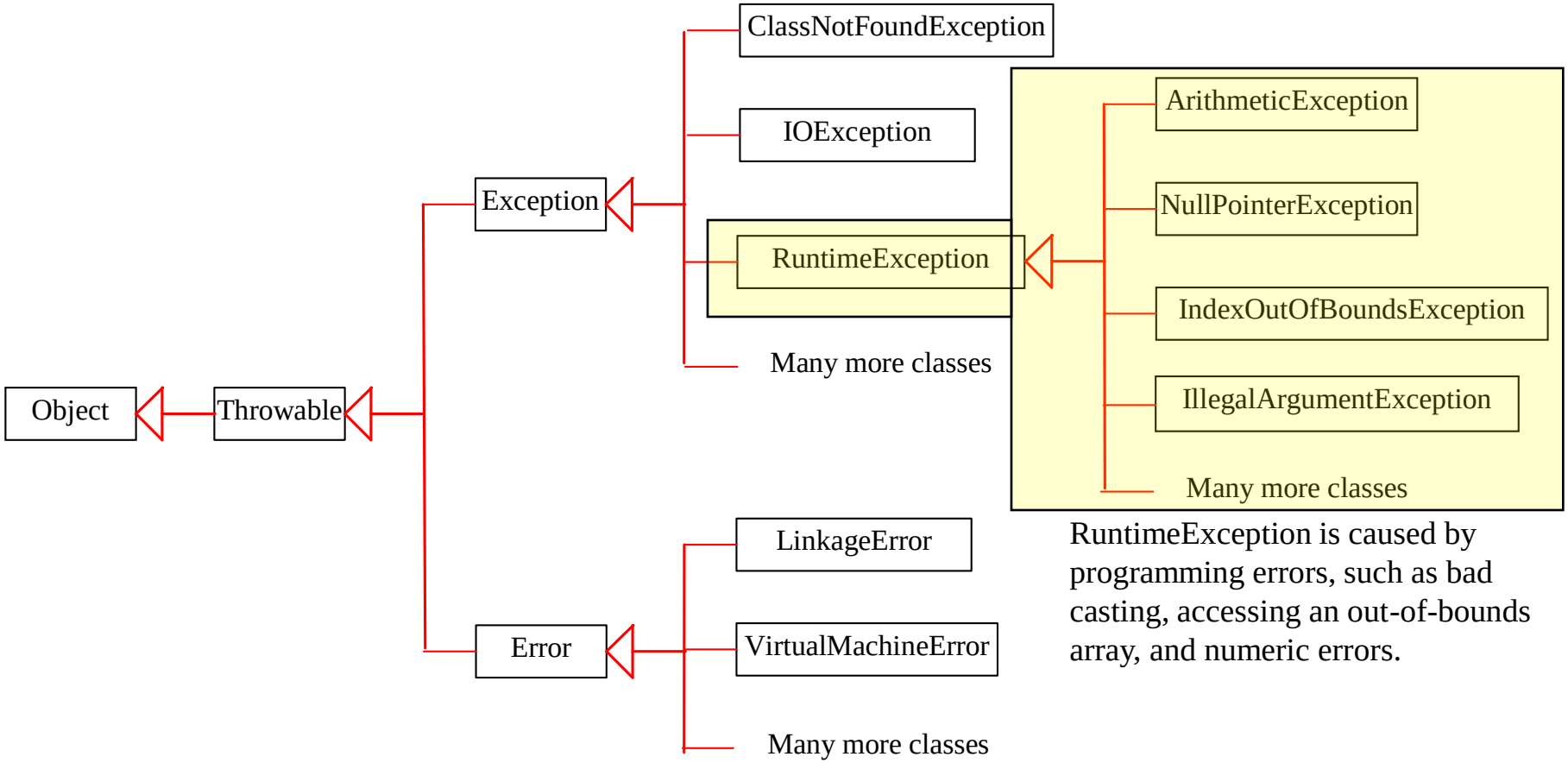
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Exceptions

Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.



Runtime Exceptions

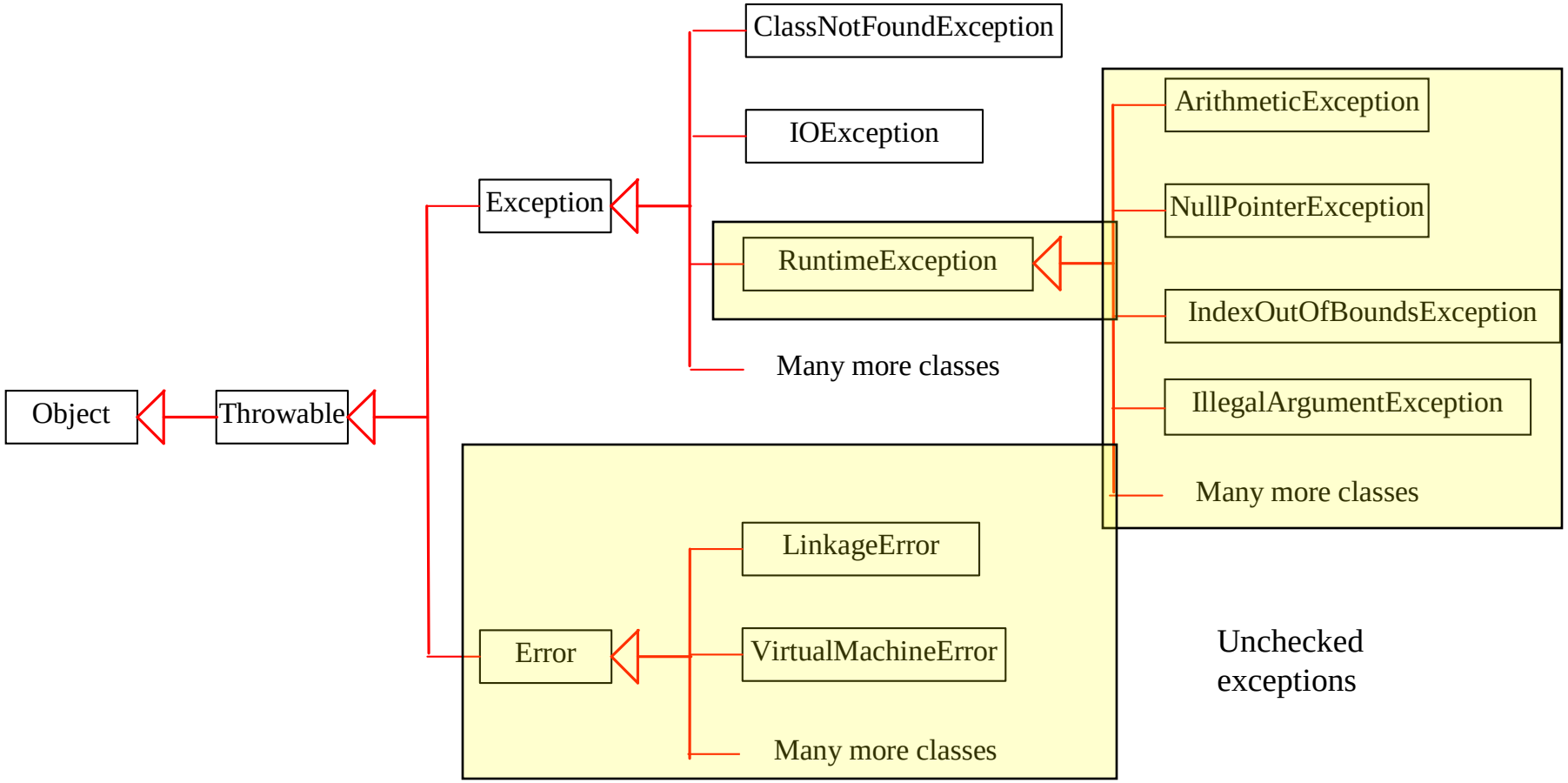


RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

Unchecked Exceptions vs. Checked Exceptions

- RuntimeException, Error and their subclasses are known as *unchecked exceptions*.
- All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

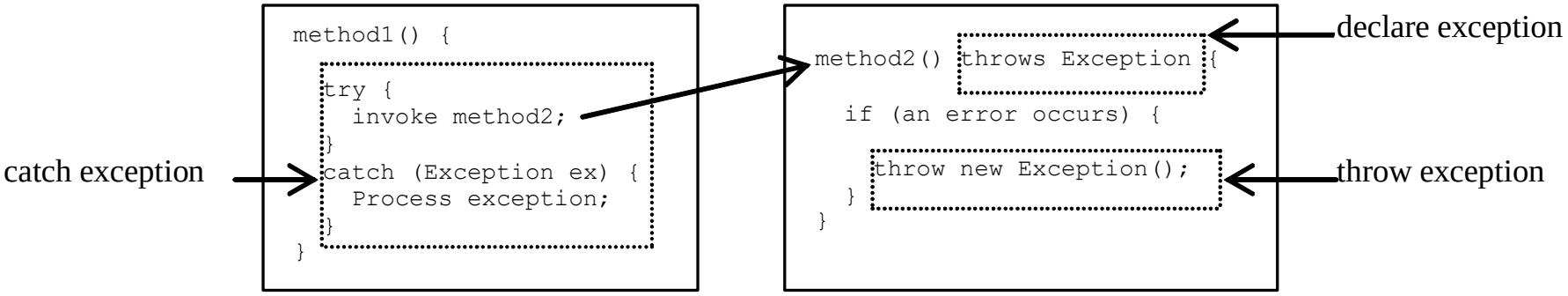
Unchecked Exceptions



Unchecked Exceptions

- In most cases, unchecked exceptions reflect programming logic errors that are not recoverable.
  - For example, a NullPointerException is thrown if you access an object through a reference variable before an object is assigned to it; an IndexOutOfBoundsException is thrown if you access an element in an array outside the bounds of the array.
  - These are the logic errors that should be corrected in the program.
    - Unchecked exceptions can occur anywhere in the program.
    - To avoid overuse of try-catch blocks, Java does not mandate you to write code to catch unchecked exceptions.

Declaring, Throwing, and Catching Exceptions



Declaring Exceptions

- Every method must state the types of checked exceptions it might throw - this is known as *declaring exceptions*:

```
public void myMethod() throws IOException

public void myMethod()throws IOException,
OtherException, …
```

Throwing Exceptions

- When the program detects an error, the program can create an instance of an appropriate exception type and throw it - known as *throwing an exception*:

```
throw new TheException();

OR

TheException ex = new TheException();
throw ex;
```

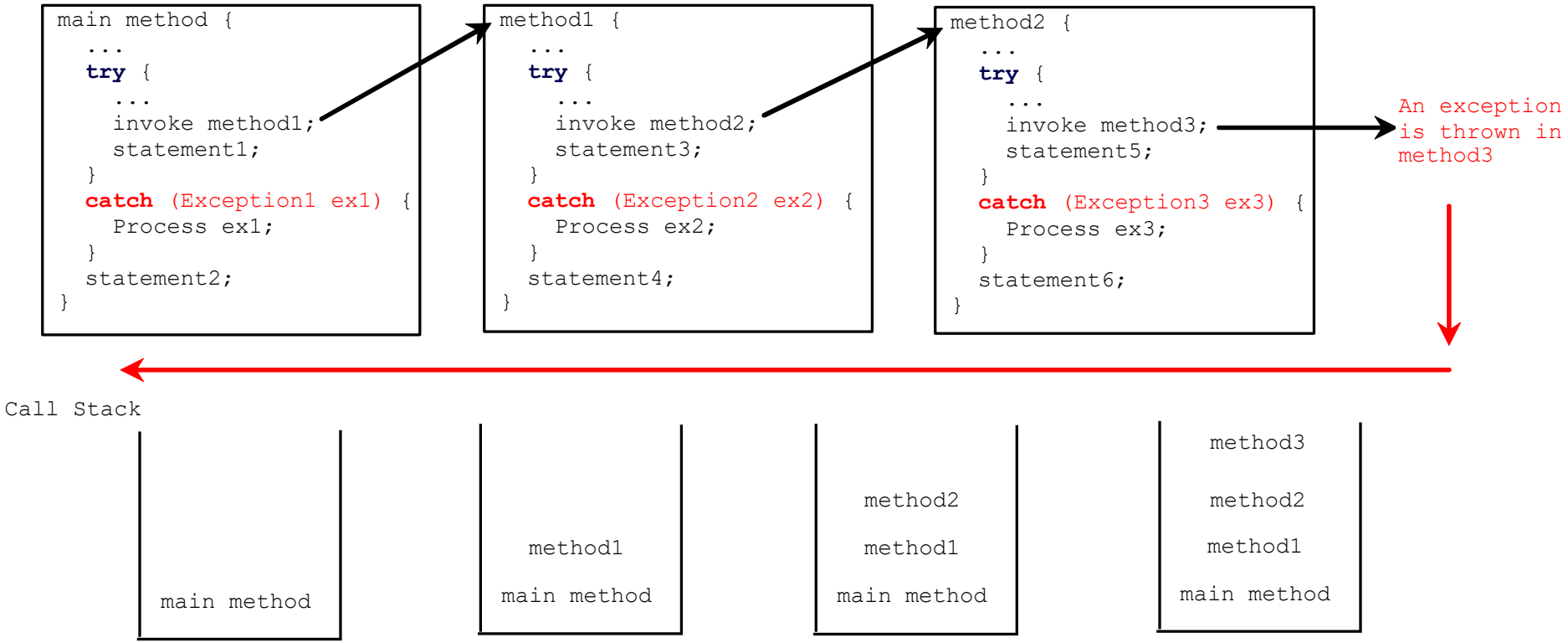
Throwing Exceptions Example

```
/** Set a new radius */
 public void setRadius(double newRadius)
     throws IllegalArgumentException {
   if (newRadius >= 0)
     radius =  newRadius;
   else
     throw new IllegalArgumentException(
       "Radius cannot be negative");
 }
```

Catching Exceptions

```
try {
  // Statements that may throw exceptions
  ...
}catch (ExceptionType1 exVar1) {
  handler for exception1;
}catch (ExceptionType2 exVar2) {
  handler for exception2;
}
...
}catch (ExceptionTypeN exVarN) {
  handler for exceptionN;
}
```

Catching Exceptions



```
public class TestCatchingExceptions {
  public static void main(String[] args) {
    try {
      method1();
    } catch(Exception e) {
      System.out.println("main");
    }
    System.out.println("main continues");
  }
  public static void method1() throws Exception {
    try {
      method2();
    }catch(RuntimeException e) {
      System.out.println("method1");
    }
    System.out.println("method1 continues");
  }
  public static void method2() throws Exception{
    try {
      method3();
    }catch(ArithmeticException e) {
      System.out.println("method2");
    }
    System.out.println("method2 continues");
  }
  public static void method3() throws Exception{
    //throw new ArithmeticException(); // method2 ...
    //throw new RuntimeException();    // method1 ...
    throw new Exception();             // main ...
  }
}
```

Catch or Declare Checked Exceptions

- Java forces you to deal with **checked exceptions**:
  - If a method declares a checked exception (i.e., an exception other than **`Error`** or **`RuntimeException`**), you must invoke it in a **`try-catch`** block or declare to throw the exception in the calling method
  - For example, suppose that method **`p1`** invokes method **`p2`** and **`p2`** may throw a checked exception (e.g., **`IOException`**), you have to write the code: (a) or (b):

```
void p1() {
  try {
    p2();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a)

```
void p1() throws IOException {

  p2();

}
```

(b)

```
public class CircleWithException {
  private double radius;
  private static int numberOfObjects = 0; // The number of the objects created

  public CircleWithException() throws IllegalArgumentException{
    this(1.0);
  }
  public CircleWithException(double newRadius) throws IllegalArgumentException{
    setRadius(newRadius);
    numberOfObjects++;
  }

  public void setRadius(double newRadius) throws IllegalArgumentException {
    if (newRadius >= 0)
      radius =  newRadius;
    else
      throw new IllegalArgumentException("Radius cannot be negative");
  }

  public static int getNumberOfObjects() { /** Return numberOfObjects */
    return numberOfObjects;
  }
```

```
  public static void main(String[] args) {
    try {
      CircleWithException c1 = new CircleWithException(5);
      CircleWithException c2 = new CircleWithException(-5);
      CircleWithException c3 = new CircleWithException(10);
    } catch (IllegalArgumentException ex) {
      System.out.println(ex);
    }
    System.out.println("Number of objects created: " +
       CircleWithException.getNumberOfObjects());
  }
}
```

Output:
Radius cannot be negative
Number of objects created: 1

The `finally` Clause

```
try {
   statements;
} catch(TheException ex) {
   handling ex;
} finally {
   finalStatements;
}
```

The finally block *always* executes when the try block exits

Useful for cleanup code:

```
}finally {
   if (out != null) {
      System.out.println("Closing PrintWriter");
      out.close();
   }
}
```

Rethrowing Exceptions

```
try {
   statements;
} catch(TheException ex) {
   …
   throw ex;
}
```

When To Use Exceptions

- Exception handling usually requires more time and resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.
  - If an exception occurs in a method:
    - If you can handle the exception in the method where it occurs, there is no need to throw it.
    - If you want the exception to be processed by its caller, you should create an exception object and throw it.

When To Use Exceptions

- We only use exceptions to deal with unexpected error conditions.
  - Do not use it to deal with simple, expected situations:

```
try {
  System.out.println(refVar.toString());
} catch (NullPointerException ex) {
  System.out.println("refVar is null");
}
```

is better to be replaced by

```
if (refVar != null)
  System.out.println(refVar.toString());
else
  System.out.println("refVar is null");
```

Defining Custom Exception Classes

- Use the exception classes in the API whenever possible.
- Define custom exception classes if the predefined classes are not sufficient.
  - Define custom exception classes by extending **`Exception`** or a subclass of **`Exception`**

Custom Exception Class Example

```
public class InvalidRadiusException extends Exception {
  private double radius;

  /** Construct an exception */
  public InvalidRadiusException(double radius) {
    super("Invalid radius " + radius);
    this.radius = radius;
  }

  /** Return the radius */
  public double getRadius() {
    return radius;
  }
}
```

Custom Exception Class Example

```
public class CircleWithInvalidRadiusException {
  private double radius;
  public CircleWithInvalidRadiusException(double newRadius)
      throws InvalidRadiusException {
    setRadius(newRadius);
    numberOfObjects++;
  }
  public void setRadius(double newRadius) throws InvalidRadiusException {
    if (newRadius >= 0)
      radius =  newRadius;
    else
      throw new InvalidRadiusException(newRadius);
  }
  public static void main(String[] args){
    try{
      CircleWithRadius c1 = new CircleWithRadius(-5);
    } catch(InvalidRadiusException e) {
      System.out.println(e.getRadius() + " is negative. No circle was created");
    }
  }
}
```

Text I/O: The `File` Class

- The **`File`** class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion.
  - The filename is a string
  - The **`File`** class is a wrapper class for the file name and its directory path

Obtaining file properties and manipulating files

java.io.File	
+File(pathname: String)	Creates a File object for the specified pathname. The pathname may be a directory or a file.
+File(parent: String, child: String)	Creates a File object for the child under the directory parent. child may be a filename or a subdirectory.
+File(parent: File, child: String)	Creates a File object for the child under the directory parent. parent is a File object. In the preceding constructor, the parent is a string.
+exists(): boolean	Returns true if the file or the directory represented by the File object exists.
+canRead(): boolean	Returns true if the file represented by the File object exists and can be read.
+canWrite(): boolean	Returns true if the file represented by the File object exists and can be written.
+isDirectory(): boolean	Returns true if the File object represents a directory.
+isFile(): boolean	Returns true if the File object represents a file.
+isAbsolute(): boolean	Returns true if the File object is created using an absolute path name.
+isHidden(): boolean	Returns true if the file represented in the File object is hidden. The exact definition of *hidden* is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period character '.'.
+getAbsolutePath(): String	Returns the complete absolute file or directory name represented by the File object.
+getCanonicalPath(): String	Returns the same as getAbsolutePath() except that it removes redundant names, such as "." and "..", from the pathname, resolves symbolic links (on Unix platforms), and converts drive letters to standard uppercase (on Win32 platforms).
+getName(): String	Returns the last name of the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getName() returns test.dat.
+getPath(): String	Returns the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getPath() returns c:\book\test.dat.
+getParent(): String	Returns the complete parent directory of the current directory or the file represented by the File object. For example, new File("c:\\book\\test.dat").getParent() returns c:\book.
+lastModified(): long	Returns the time that the file was last modified.
+delete(): boolean	Deletes this file. The method returns true if the deletion succeeds.
+renameTo(dest: File): boolean	Renames this file. The method returns true if the operation succeeds.

Text I/O

- A **`File`** object encapsulates the properties of a file or a path, but ***does not contain the methods for reading/writing data*** from/to a file.
  - In order to perform Input and Output (I/O), you need to create objects using appropriate Java I/O classes: **`PrintWriter`** and **`Scanner`**

Writing Data Using PrintWriter

java.io.PrintWriter	
+PrintWriter(file: File)	Creates a PrintWriter for the specified file.
+print(s: String): void	Writes a string.
+print(c: char): void	Writes a character.
+print(cArray: char[]): void	Writes an array of character.
+print(i: int): void	Writes an int value.
+print(l: long): void	Writes a long value.
+print(f: float): void	Writes a float value.
+print(d: double): void	Writes a double value.
+print(b: boolean): void	Writes a boolean value.
Also contains the overloaded println methods.	A println method acts like a print method; additionally it prints a line separator. The line separator string is defined by the system. It is \r\n on Windows and \n on Unix.
Also contains the overloaded printf methods.	The printf method was introduced in §3.6, "Formatting Console Output and Strings."

```
public class WriteData {
  public static void main(String[] args)
                throws Exception {
    java.io.File file = new java.io.File("scores.txt");
    if (file.exists()) {
      System.out.println("File already exists");
      System.exit(0);
    }
    // Create the file
    java.io.PrintWriter output = new
                  java.io.PrintWriter(file);
    // Write output to the file
    output.println("mary 100");
    output.println("john 90");
    // Close the file
    output.close();
  }
}
```

Reading Data Using Scanner

java.util.Scanner	
+Scanner(source: File)	Creates a Scanner that produces values scanned from the specified file.
+Scanner(source: String)	Creates a Scanner that produces values scanned from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has another token in its input.
+next(): String	Returns next token as a string.
+nextByte(): byte	Returns next token as a byte.
+nextShort(): short	Returns next token as a short.
+nextInt(): int	Returns next token as an int.
+nextLong(): long	Returns next token as a long.
+nextFloat(): float	Returns next token as a float.
+nextDouble(): double	Returns next token as a double.
+useDelimiter(pattern: String): Scanner	Sets this scanner's delimiting pattern.

```
import java.util.Scanner;
public class ReadData {
  public static void main(String[] args)throws Exception{
    // Create a File instance
    java.io.File file = new java.io.File("scores.txt");
    // Create a Scanner for the file
    Scanner input = new Scanner(file);
    // Read data from a file
    while (input.hasNext()) {
      String user = input.next();
      int score = input.nextInt();
      System.out.println(user + " " + score);
    }
    // Close the file
    input.close();
  }
}
```