Chapter 12 Exception Handling and Text IO

Motivations

When a program runs into a runtime error, the program terminates abnormally. How can you handle the runtime error so that the program can continue to run or terminate gracefully? This is the subject we will introduce in this chapter.

Objectives

- To get an overview of exceptions and exception handling (§12.2).
- To explore the advantages of using exception handling (§12.2).
- To distinguish exception types: **Error** (fatal) vs. **Exception** (nonfatal) and checked vs. unchecked (§12.3).
- To declare exceptions in a method header (§12.4.1).
- To throw exceptions in a method (§12.4.2).
- To write a **try-catch** block to handle exceptions (§12.4.3).
- To explain how an exception is propagated (§12.4.3).
- To obtain information from an exception object (§12.4.4).
- To develop applications with exception handling (§12.4.5).
- To use the **finally** clause in a **try-catch** block (§12.5).
- To use exceptions only for unexpected errors (§12.6).
- To **rethrow exceptions** in a **catch** block (§12.7).
- To create chained exceptions (§12.8).
- To define custom exception classes (§12.9).
- To discover file/directory properties, to delete and rename files/directories, and to create directories using the **File** class (§12.10).
- To write data to a file using the **PrintWriter** class (§12.11.1).
- To use try-with-resources to ensure that the resources are closed automatically (§12.11.2).
- To read data from a file using the **Scanner** class (§12.11.3).
- To understand how data is read using a **Scanner** (§12.11.4).
- To develop a program that replaces text in a file (§12.11.5).
- To read data from the Web (§12.12).
- To develop a Web crawler (§12.13).

Exception Handling

- The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.
- An exception is an event that **occurs during the execution of a program** that disrupts the normal flow of instructions.
- ***Run time error*** *occurs during the execution of a program. In contrast, compile-time errors occur while a program is being compiled. Runtime errors indicate bugs in the program or problems that the designers had anticipated but could do nothing about. For example, running out of memory will often cause a runtime error.*

Difference between error & exception

- **Errors** indicate serious problems and abnormal conditions that most applications should not try to handle.
- Error defines problems that are not expected to be caught under normal circumstances by program. For example memory error, hardware error, JVM error etc. **Exceptions** are conditions within the code.
- A developer can handle such conditions and take necessary corrective actions. Few examples – DivideByZero exception, NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException

Try Catch in Java – Exception handling

- A **try** block is always followed by a **catch** block, which *handles the exception* that occurs in associated try block

Syntax of try block

```
try{
   //statements that may cause an exception
}
```

Syntax of try catch in java

```
try
{
     //statements that may cause an exception
}
catch (exception(type) e(object))
{
     //error handling code
}
```

A catch block must be associated with a try block

```
class Nest{
   public static void main(String args[]){
    //Parent try block
     try{
      //Child try block1
         try{
         //try-catch block inside another try block
            System.out.println("Inside block1");
            int b =45/0;
            System.out.println(b);
         }
         catch(ArithmeticException e1){
          //Exception Message
            System.out.println("Exception: e1");
         }
        //Child try block2
         try{
          //try-catch block inside another try block
            System.out.println("Inside block2");
            int b =45/0;
            System.out.println(b);
         }
         catch(ArrayIndexOutOfBoundsException e2){
          //Exception Message
            System.out.println("Exception: e2");
             }
        System.out.println("Just other statement");
    }
    catch(ArithmeticException e3){  //Catch of Main(parent) try
block
          //Exception Message
       System.out.println("Arithmetic Exception");
         System.out.println("Inside parent try catch block");
    }
    catch(ArrayIndexOutOfBoundsException e4){
      System.out.println("ArrayIndexOutOfBoundsException");
         System.out.println("Inside parent try catch block");
    }
    catch(Exception e5){
      System.out.println("Exception");
         System.out.println("Inside parent try catch block");
     }
     System.out.println("Next statement..");
   }
}
```

Flow of try catch block program can also contain **nested try-catch-finally blocks.**

```
----jGRASP exec: java Nest

Inside block1
Exception: e1
Inside block2
Arithmetic Exception
Inside parent try catch block
Next statement..

----jGRASP: operation complete.
```

catch, try, throw

The Catch or Specify Requirement

How to ***catch*** and **try** handle exceptions.

Also, the **throw** statement and

the **Throwable** class and its subclasses.

The try and catch Block - Weblink

Java Document by Oracle Weblink

try and catch question and answer -- Weblink

Exception-Handling Overview

Show runtime error

Quotient

```
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));
  }
}
```

Fix it using an if statement

QuotientWithIf

```
import java.util.Scanner;

public class QuotientWithIf {
  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();

    if (number2 != 0)
      System.out.println(number1 + " / " + number2 +
" is " +
        (number1 / number2));
    else
      System.out.println("Divisor cannot be zero ");
  }
}
```

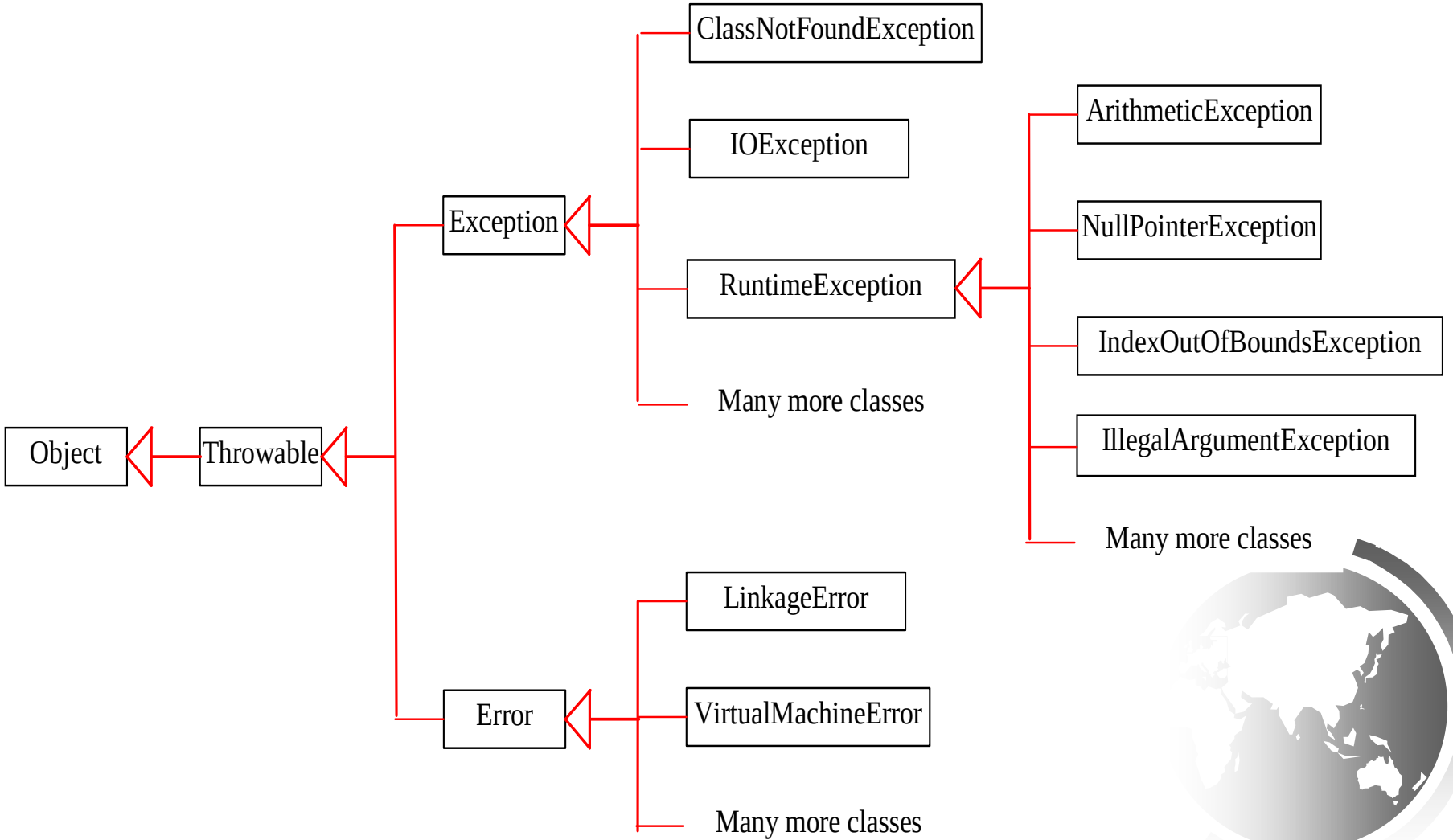
With a method

QuotientWithMethod

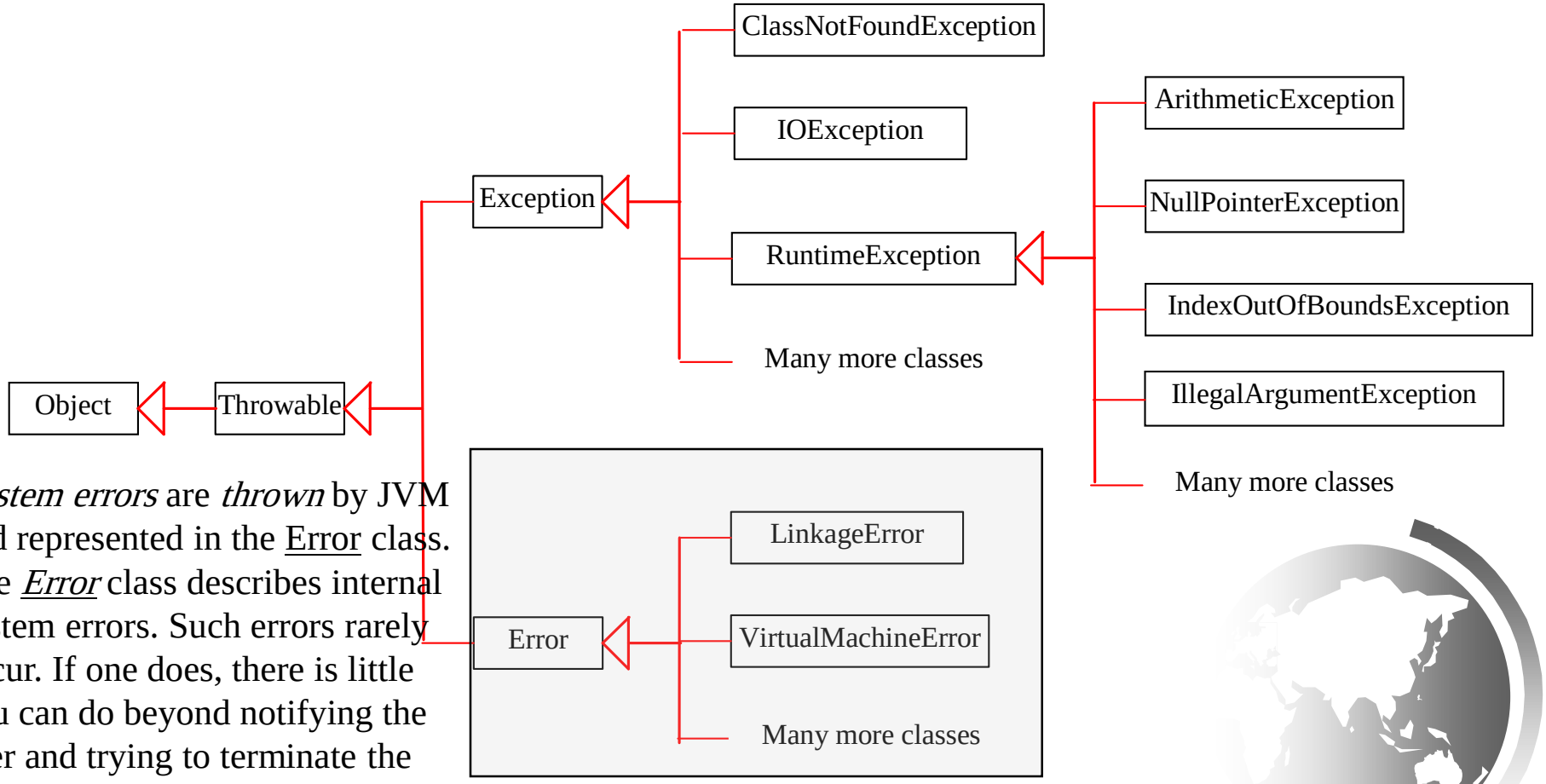
```
import java.util.Scanner;

public class QuotientWithMethod {
  public static int quotient(int number1,
int number2) {
    if (number2 == 0) {
      System.out.println("Divisor cannot be
zero");
      System.exit(1);
    }

    return number1 / number2;
```

```
----jGRASP exec: java QuotientWithMethod

Enter two integers: 5 3
5 / 3 is 1

----jGRASP: operation complete.

----jGRASP exec: java QuotientWithMethod

Enter two integers: 5 0
Divisor cannot be zero

----jGRASP wedge2: exit code for process is 1
----jGRASP: operation complete.
```

Exception Types



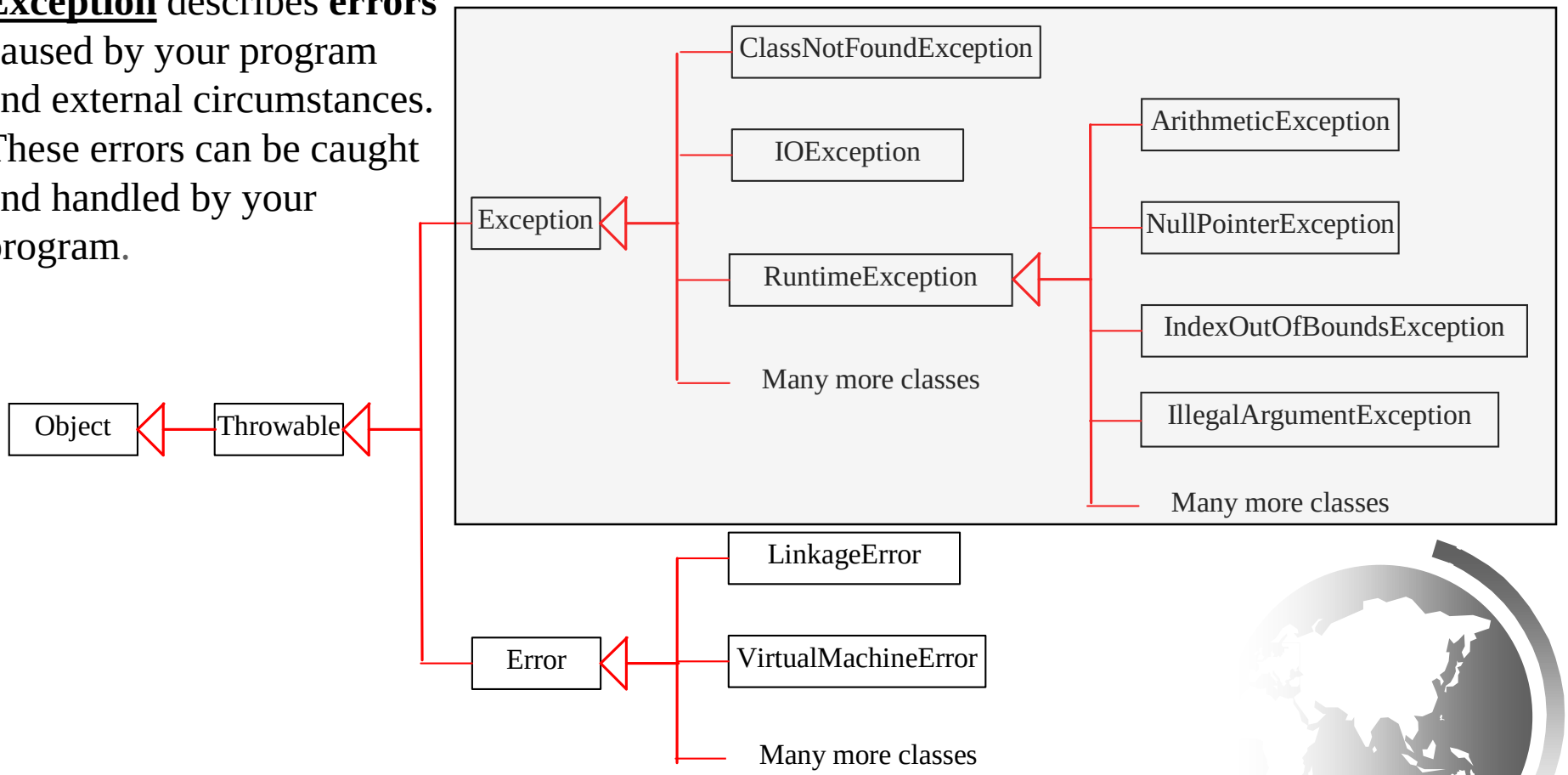
System Errors



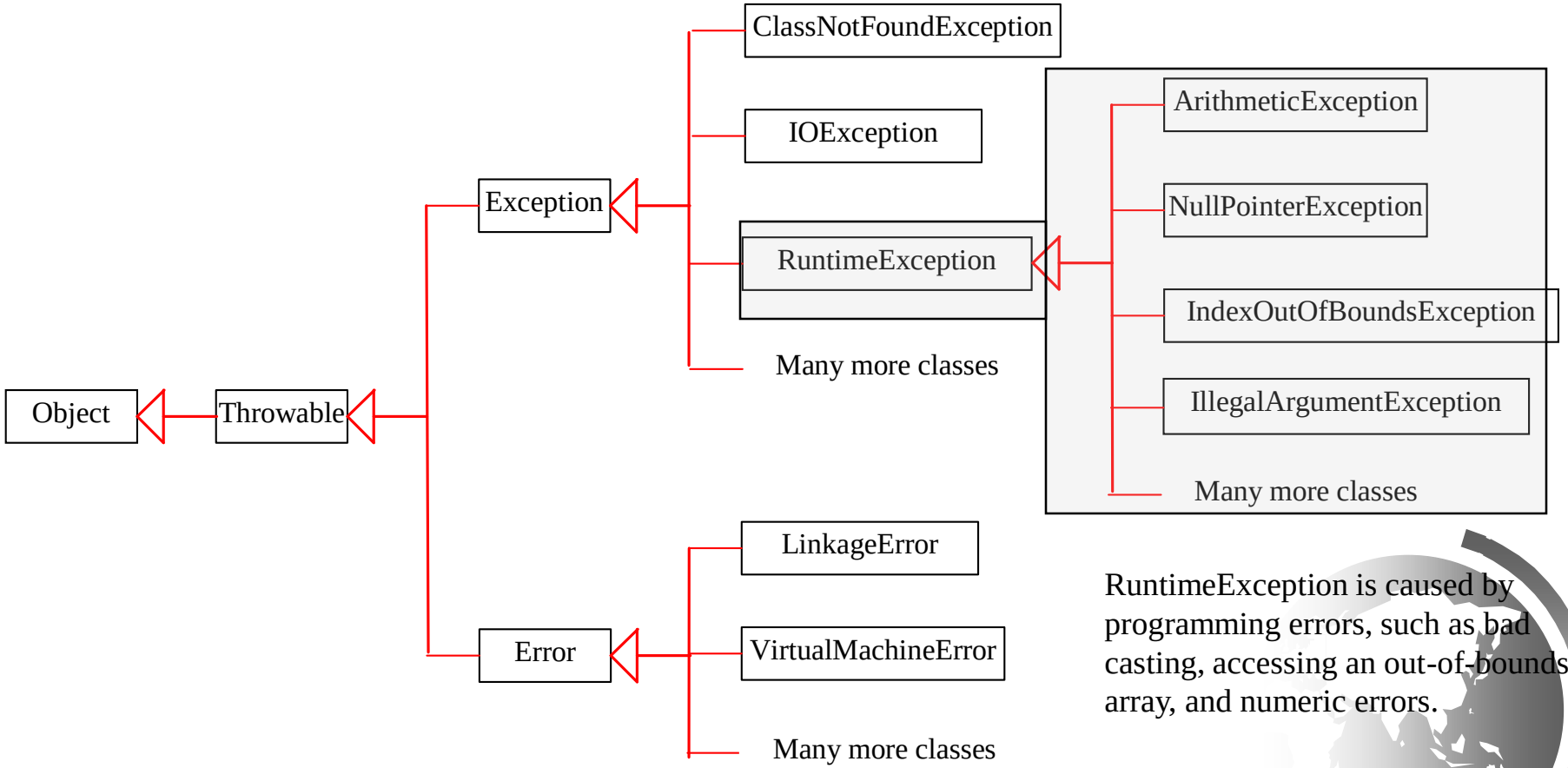
System errors are *thrown* by JVM and represented in the Error class. The *Error* class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Exceptions

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



Runtime Exceptions

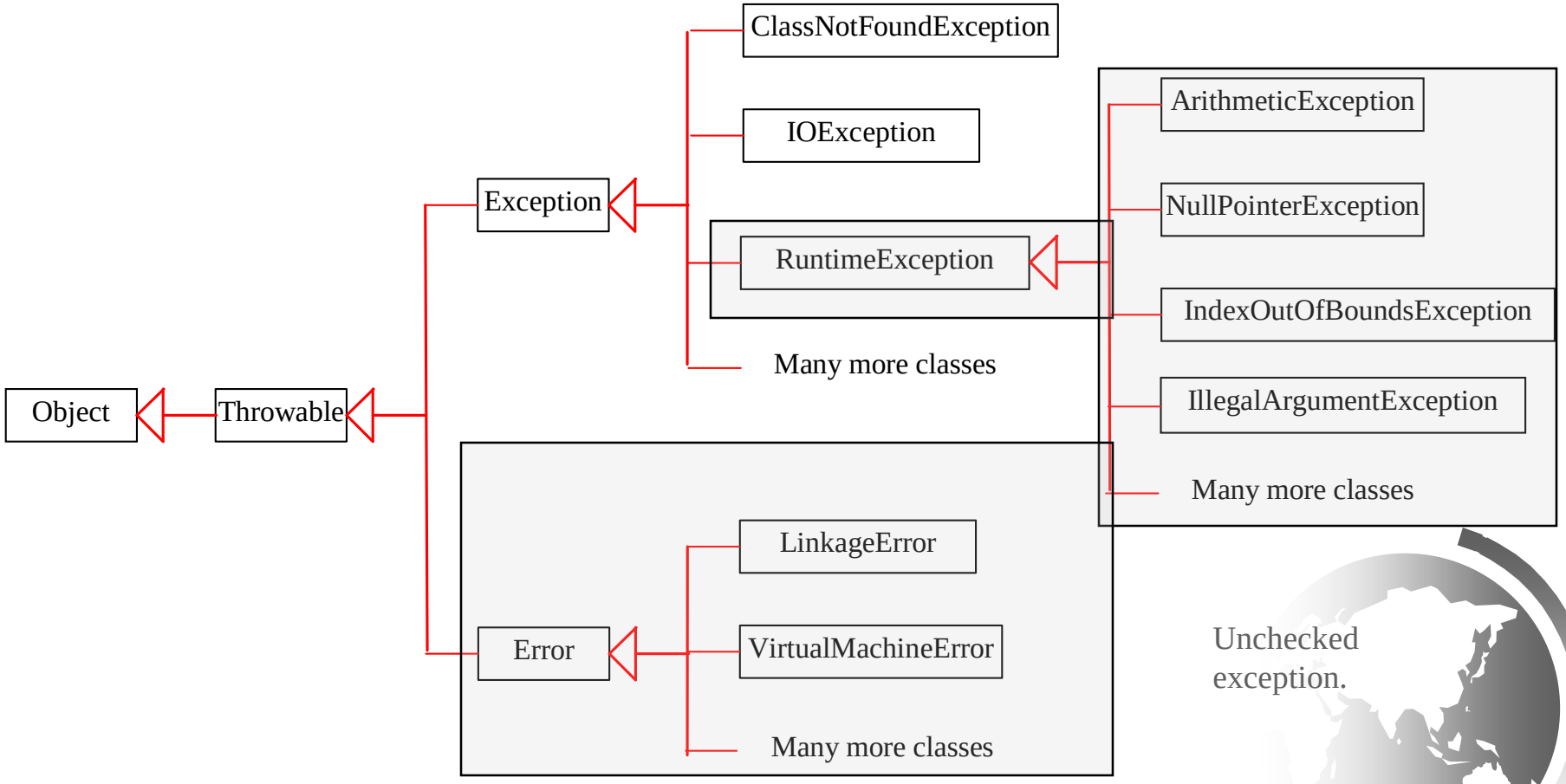


RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

Checked Exceptions vs. Unchecked Exceptions

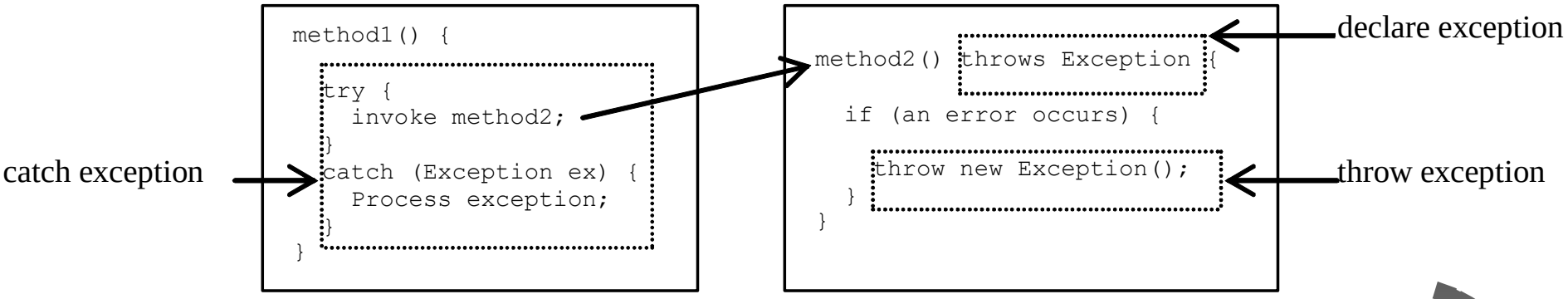
RuntimeException, Error and their subclasses are known as *unchecked exceptions*. All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

Unchecked Exceptions



Unchecked exception.

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. This is known as *throwing an exception*. Here is an example,

```
throw new TheException();
```

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

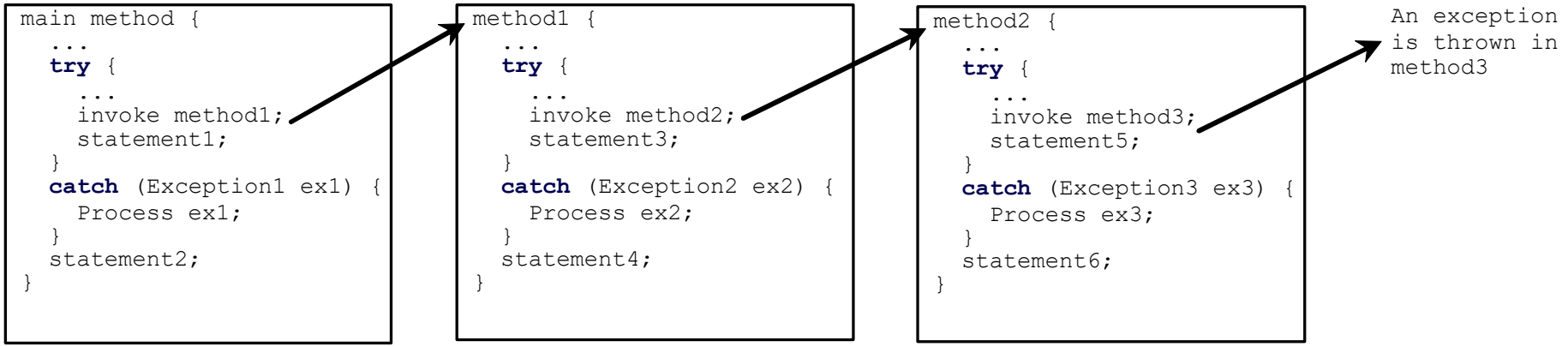
Throwing Exceptions Example

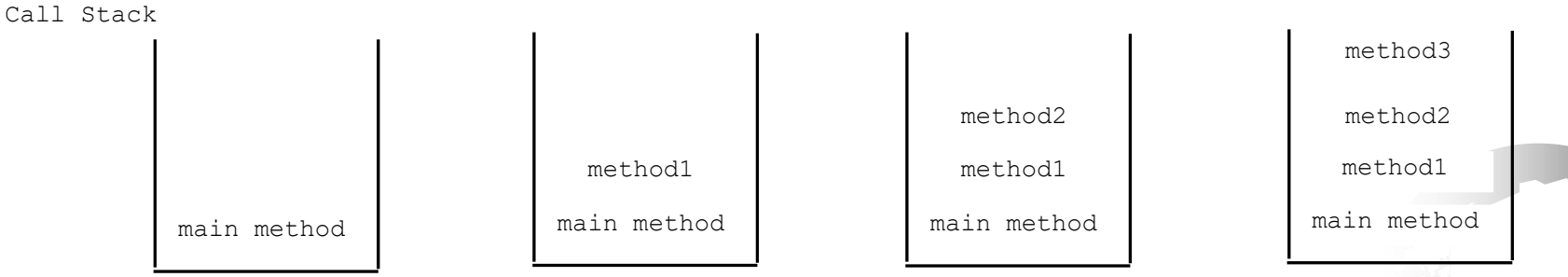
```
   /** Set a new radius */
public void setRadius(double newRadius)
    throws IllegalArgumentException {
  if (newRadius >= 0)
    radius =  newRadius;
  else
    throw new IllegalArgumentException(
      "Radius cannot be negative");
}
```

Catching Exceptions

```
try {
  statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVar3) {
  handler for exceptionN;
}
```

Catching Exceptions





Catch or Declare Checked Exceptions

Suppose p2 is defined as follows:

```
void p2() throws IOException {
  if (a file does not exist) {
     throw new IOException("File does not exist");
  }

  ...
}
```

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 as shown in (a) or (b).

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

(a)

```
void p1() throws IOException {

  p2();

}
```

(b)

Example: **Declaring**, **Throwing**, and **Catching Exceptions**

- Objective: This example demonstrates declaring, throwing, and catching exceptions by modifying the setRadius method in the Circle class defined in Chapter 8. The new setRadius method throws an exception if radius is negative.

TestCircleWithException

CircleWithException

Run

TestCircleWithException

```
public class TestCircleWithException {
 public static void main(String[] args) {
   try {
    CircleWithException c1 = new CircleWithException(5);
    CircleWithException c2 = new CircleWithException(-5);
    CircleWithException c3 = new CircleWithException(0);
   }
   catch (IllegalArgumentException ex) {
    System.out.println(ex);
   }

   System.out.println("Number of objects created: " +
    CircleWithException.getNumberOfObjects());
 }
}
```

```
----jGRASP exec: java TestCircleWithException

java.lang.IllegalArgumentException: Radius cannot be negative
Number of objects created: 1

----jGRASP: operation complete.
```

CircleWithException

```
public class CircleWithException {
  /** The radius of the circle */
  private double radius;
  /** The number of the objects created */
  private static int numberOfObjects = 0;
  /** Construct a circle with radius 1 */
  public CircleWithException() {
    this(1.0);
  }
  /** Construct a circle with a specified radius */
  public CircleWithException(double newRadius) {
    setRadius(newRadius);
    numberOfObjects++;
  }
  /** Return radius */
  public double getRadius() {
    return radius;
  }
  /** Set a new radius */
  public void setRadius(double newRadius)
      throws IllegalArgumentException {
    if (newRadius >= 0)
      radius =  newRadius;
    else
      throw new IllegalArgumentException(
       "Radius cannot be negative");
  }
  /** Return numberOfObjects */
  public static int getNumberOfObjects() {
    return numberOfObjects;
  }
  /** Return the area of this circle */
  public double findArea() {
    return radius * radius * 3.14159;
  }
}
```

Rethrowing Exceptions

```
try {
  statements;
}
catch(TheException ex) {
  perform operations before exits;
  throw ex;
}
```

The `finally` Clause

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

animation

Trace a Program Execution

Suppose no exceptions in the statements

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

Next statement;
```

animation

Trace a Program Execution

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

Next statement;
```

The final block is always executed

animation

Trace a Program Execution

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

Next statement;
```

Next statement in the method is executed

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Suppose an exception of type Exception1 is thrown in statement2

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The exception is handled.

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed.

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The next statement in the method is now executed.

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

statement2 throws an exception of type Exception2.

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Handling exception

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Execute the final block

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Rethrow the exception and control is transferred to the caller

When to Throw Exceptions

- An exception occurs in a method.
- **If you want the exception to be processed by its caller**,
  - Then **you should create an exception object and throw** it.
  - **If you can handle the exception in the method where it occurs, there is no need to throw it.**

When to Use Exceptions

When should you use the try-catch block in the code? You should use it to deal with unexpected error conditions. Do not use it to deal with simple, expected situations. For example, the following code

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

When to Use Exceptions

is better to be replaced by

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

Custom Exception Class Example

In Listing 13.8, the setRadius method throws an exception if the radius is negative. Suppose you wish to pass the radius to the handler, you have to create a custom exception class.

InvalidRadiusException

CircleWithRadiusException

TestCircleWithRadiusException

Run

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 file

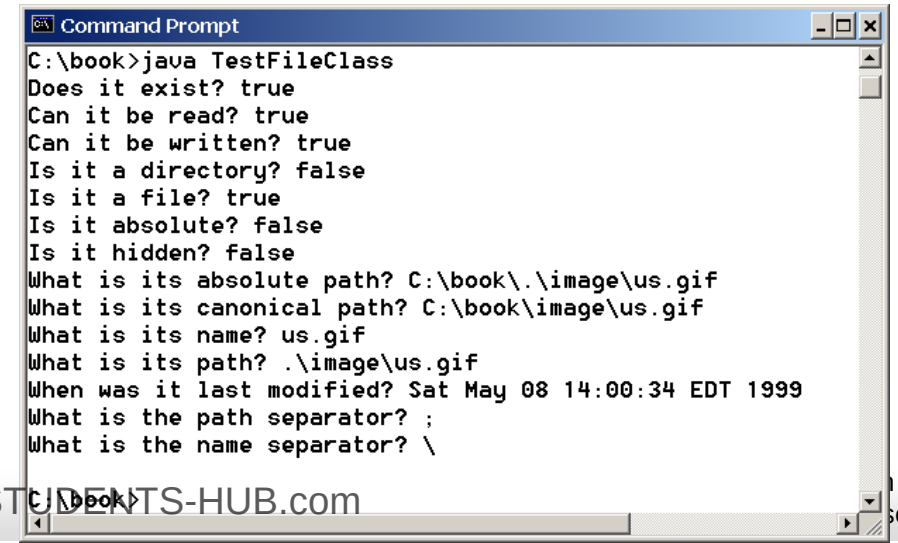
java.io.File	
`+File(pathname: String)`	Creates a `File` object for the specified path name. The path name may be a directory or a file.
`+File(parent: String, child: String)`	Creates a `File` object for the child under the directory parent. The child may be a file name or a subdirectory.
`+File(parent: File, child: String)`	Creates a `File` object for the child under the directory parent. The 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 path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).
`+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.
`+length(): long`	Returns the size of the file, or 0 if it does not exist or if it is a directory.
`+listFile(): File[]`	Returns the files under the directory for a directory `File` object.
`+delete(): boolean`	Deletes the file or directory represented by this `File` object.The method returns true if the deletion succeeds.
`+renameTo(dest: File): boolean`	Renames the file or directory represented by this `File` object to the specified name represented in dest. The method returns true if the operation succeeds.
`+mkdir(): boolean`	Creates a directory represented in this `File` object. Returns true if the the directory is created successfully.
`+mkdirs(): boolean`	Same as `mkdir()` except that it creates directory along with its parent directories if the parent directories do not exist.

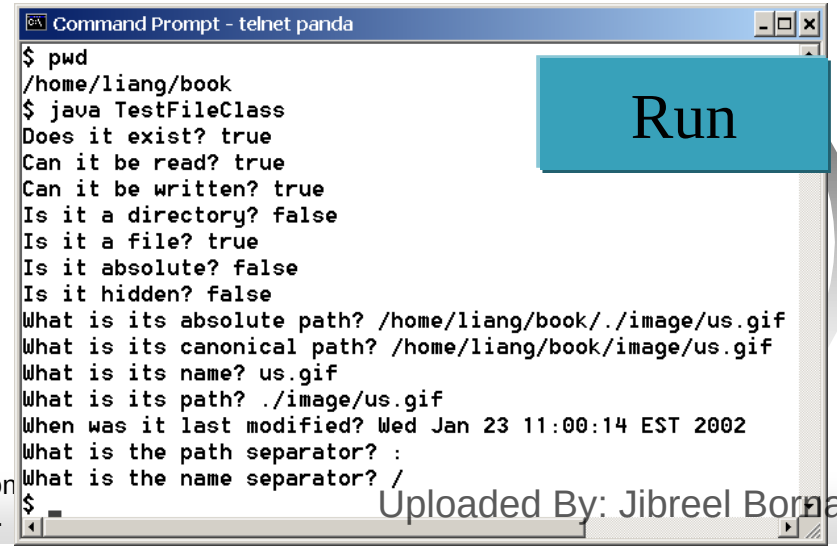
Problem: Explore File Properties

Objective: Write a program that demonstrates how to create files in a platform-independent way and use the methods in the File class to obtain their properties. The following figures show a sample run of the program on Windows and on Unix.

TestFileClass

```
public class TestFileClass {
  public static void main(String[] args) {
    java.io.File file = new
java.io.File("image/us.gif");
    System.out.println("Does it exist? " +
file.exists());
    System.out.println("The file has " +
file.length() + " bytes");
    System.out.println("Can it be read? " +
file.canRead());
    System.out.println("Can it be written? " +
file.canWrite());
    System.out.println("Is it a directory? " +
file.isDirectory());
    System.out.println("Is it a file? " +
file.isFile());
    System.out.println("Is it absolute? " +
file.isAbsolute());
    System.out.println("Is it hidden? " +
file.isHidden());
    System.out.println("Absolute path is " +
      file.getAbsolutePath());
    System.out.println("Last modified on " +
      new java.util.Date(file.lastModified()));
  }
}
```





Run

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 I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file. This section introduces how to read/write strings and numeric values from/to a text file using the Scanner and PrintWriter classes.

Writing Data Using PrintWriter

java.io.PrintWriter	
+PrintWriter(filename: String)	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."

WriteData

Run

WriteData

```
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 a file
    java.io.PrintWriter output = new java.io.PrintWriter(file);

  /* you can create PrintWriter objects for writing text to any file using
  print, println, and printf **/
    // Write formatted output to the file
  output.print("John T Smith ");
  output.println(90);
  output.print("Eric K Jones ");
  output.println(85);

  // Close the file
  output.close();
  }
}
```

```
----jGRASP exec: java WriteData

File already exists

----jGRASP: operation complete.
```

Reading Data Using Scanner

java.util.Scanner	
+Scanner(source: File)	Creates a Scanner object to read data from the specified file.
+Scanner(source: String)	Creates a Scanner object to read data 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.

ReadData

Run

Problem: Replacing Text

Write a class named ReplaceText that replaces a string in a text file with a new string. The filename and strings are passed as command-line arguments as follows:

```
java ReplaceText sourceFile targetFile oldString newString
```

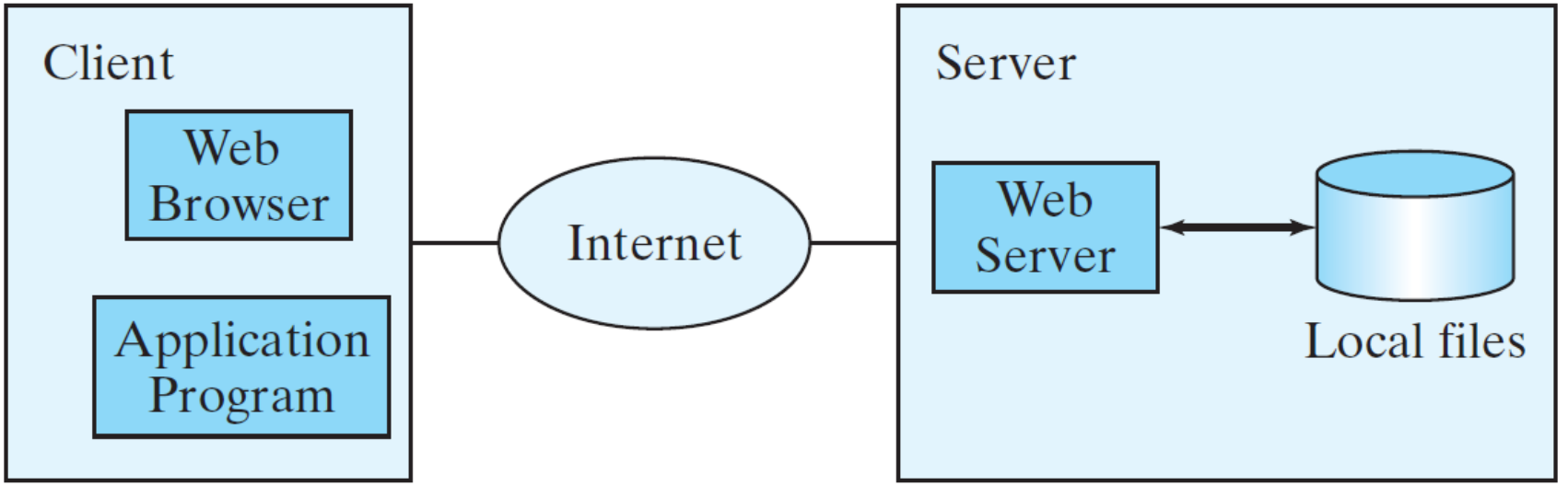
For example, invoking

```
java ReplaceText FormatString.java t.txt StringBuilder StringBuffer
```

replaces all the occurrences of StringBuilder by StringBuffer in FormatString.java and saves the new file in t.txt.

Reading Data from the Web

Just like you can read data from a file on your computer, you can read data from a file on the Web.



Reading Data from the Web

URL url = **new** URL("**www.google.com/index.html**");

After a **URL** object is created, you can use the **openStream()** method defined in the **URL** class to open an input stream and use this stream to create a **Scanner** object as follows:

Scanner input = **new** Scanner(url.openStream());