Exception Handling and Text IO

Exception-Handling Overview

Show runtime error

```java
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: 3 0 ↵Enter
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Quotient.main(Quotient.java:11)
```

Exception-Handling Overview

Fix it using if statement

```
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 ");
  }
}
```

```
Enter two integers: 5 0 ↵Enter
Divisor cannot be zero
```

Exception-Handling Overview

With a method

```java
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;
  }

  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);
  }
}
```

```
Enter two integers: 5 0 ↵Enter
Divisor cannot be zero
```

Exception Advantages

```java
import java.util.Scanner;

public class QuotientWithException {
  public static int quotient(int number1, int number2) {
    if (number2 == 0)
      throw new ArithmeticException("Divisor cannot be zero");

    return number1 / number2;
  }

  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 (ArithmeticException ex) {
      System.out.println("Exception: an integer " +
        "cannot be divided by zero ");
    }

    System.out.println("Execution continues ...");
  }
}
```

If an Arithmetic Exception occurs

```
Enter two integers: 5 0 ↵Enter
Exception: an integer cannot be divided by zero
Execution continues ...
```

Exception Advantages

* Now you see the *advantages* of using exception handling.

  - It enables a method to throw an exception to its caller.

  - Without this capability, a method must handle the exception or terminate the program.

* The key benefit of exception handling is separating:

  - the detection of an error (done in a called method)

from

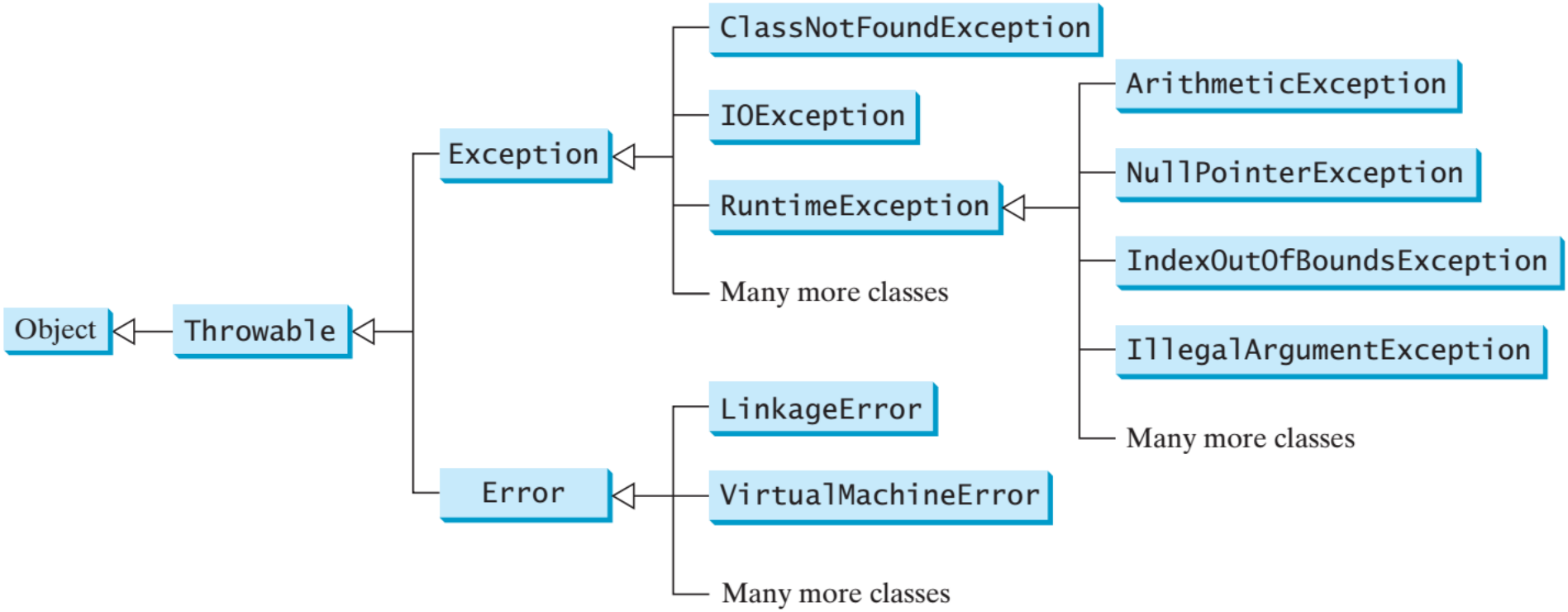
  - the handling of an error (done in the calling method).

Example: library methods

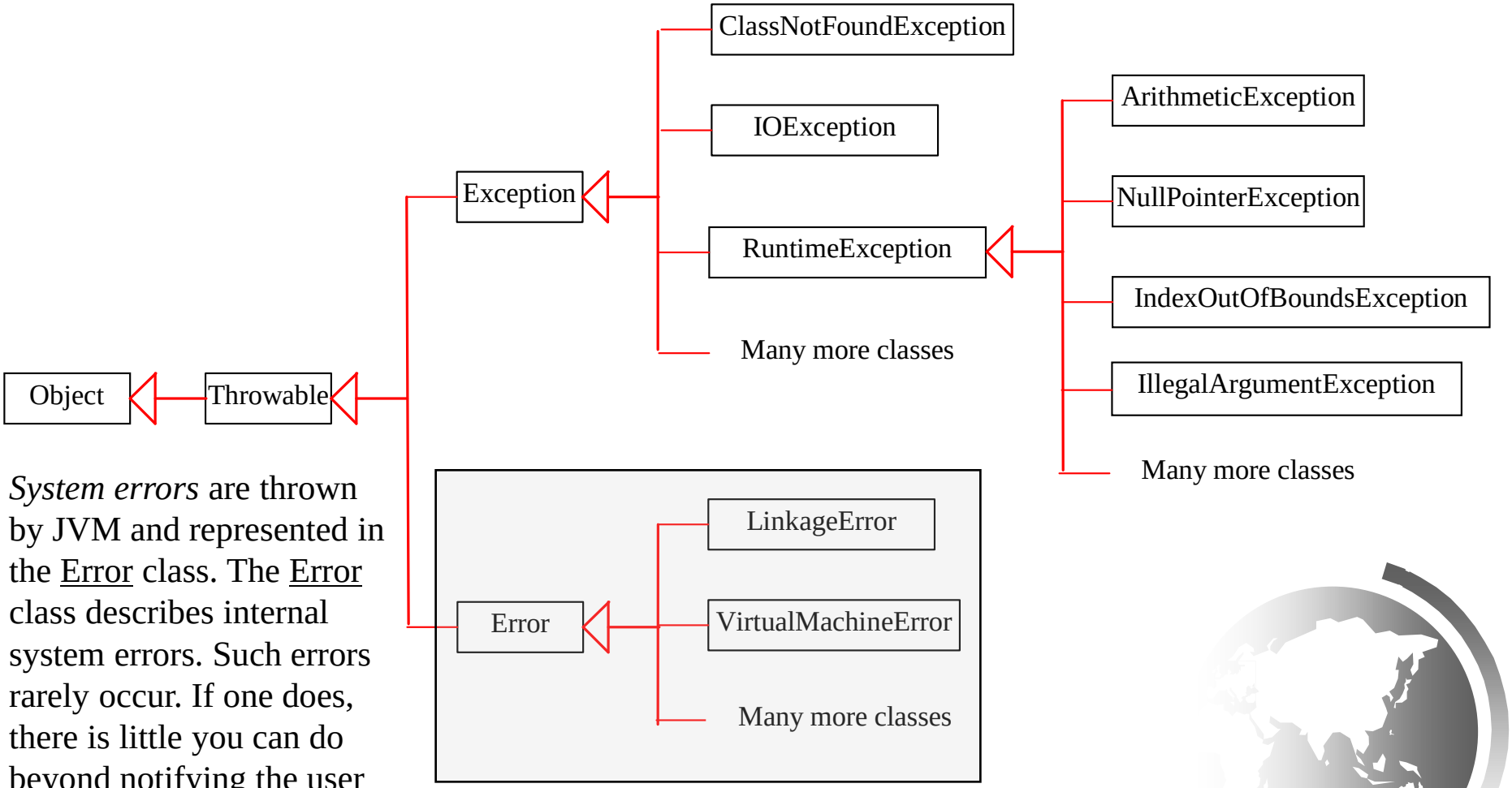
6

Exception Types

* Exceptions are objects, and objects are defined using classes.
* The root class for exceptions is java.lang.Throwable.



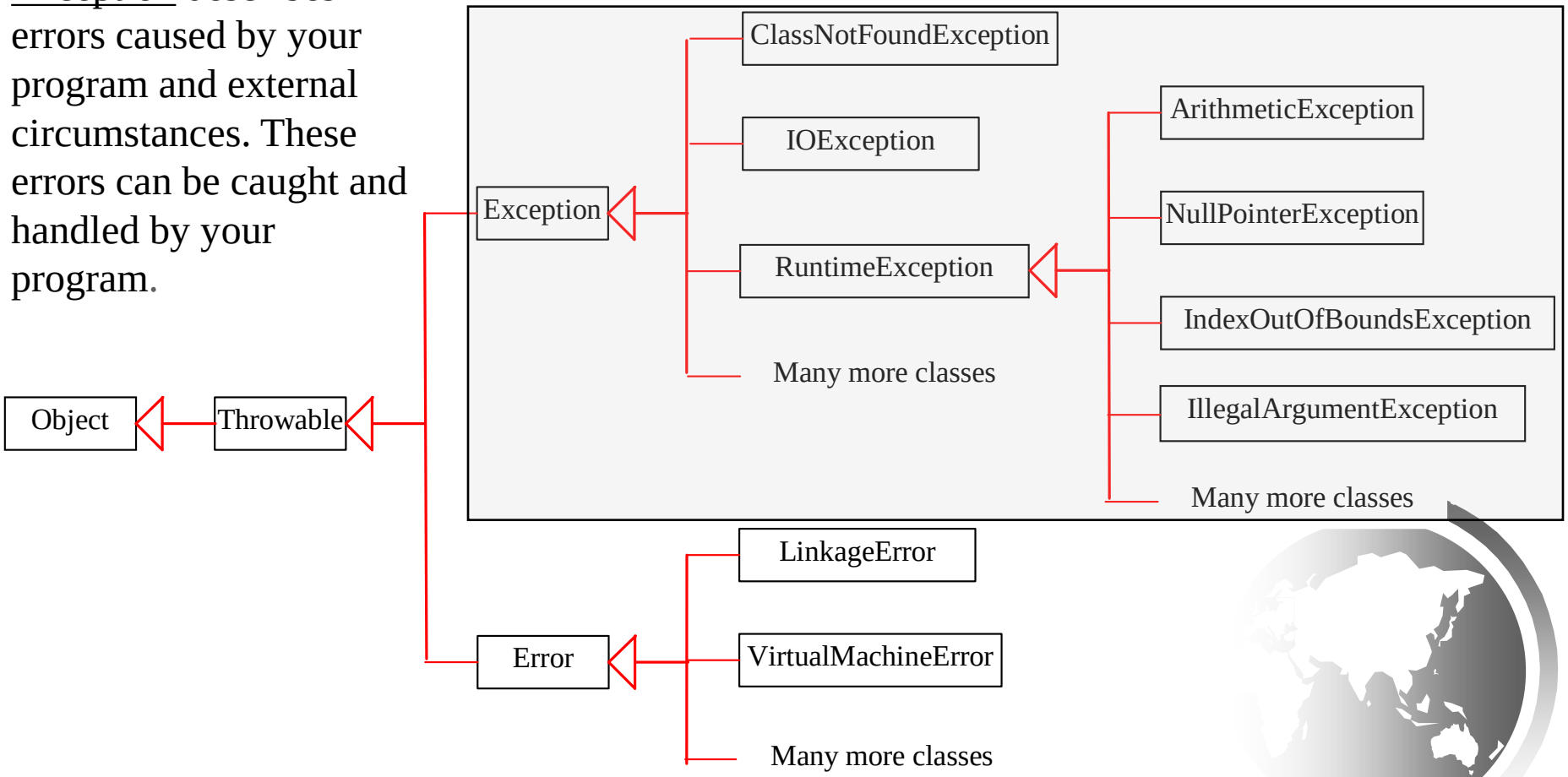
System Errors



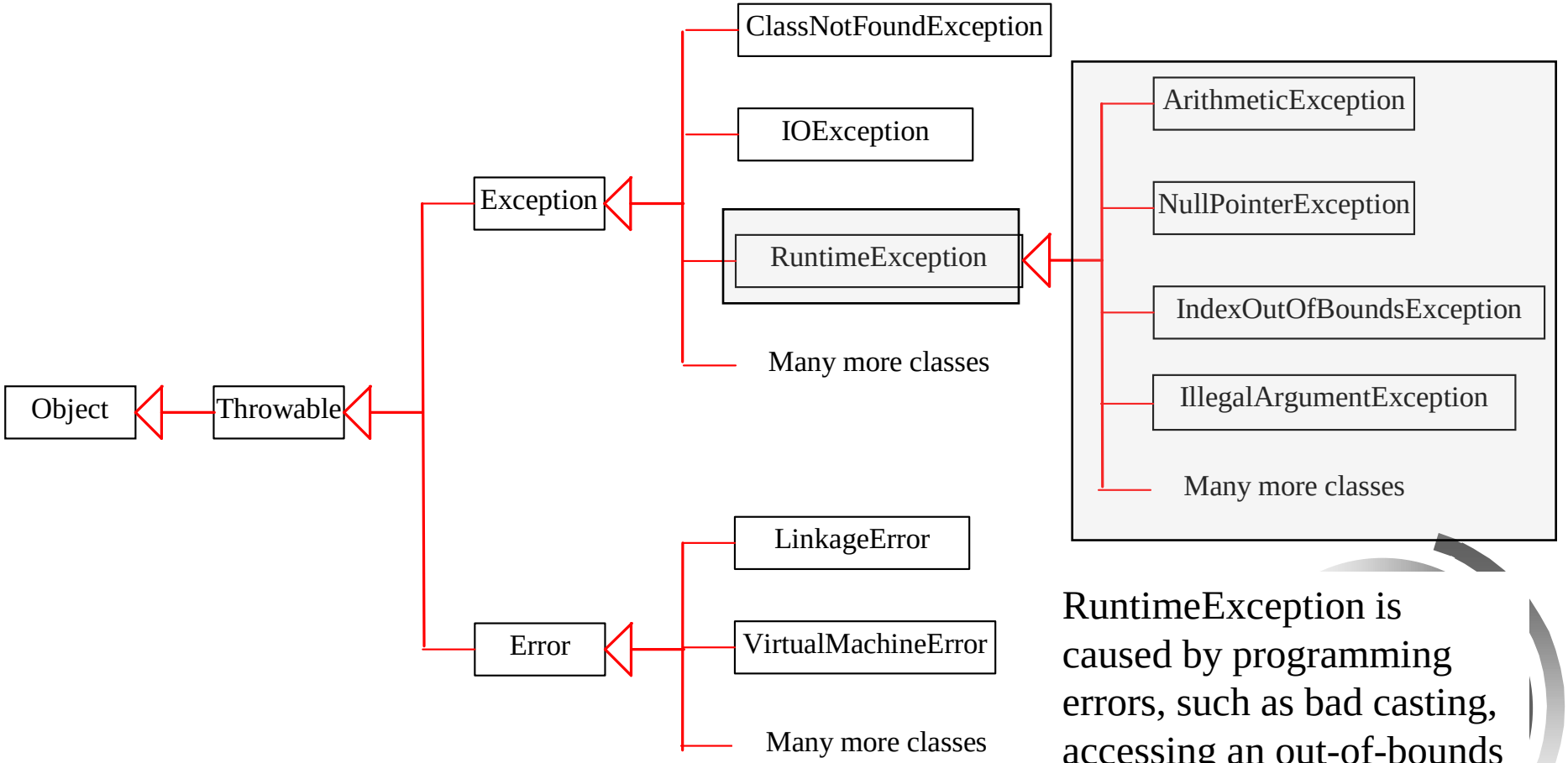
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

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*.

- usually, unchecked exceptions reflect programming logic errors that are not recoverable, can occur anywhere,

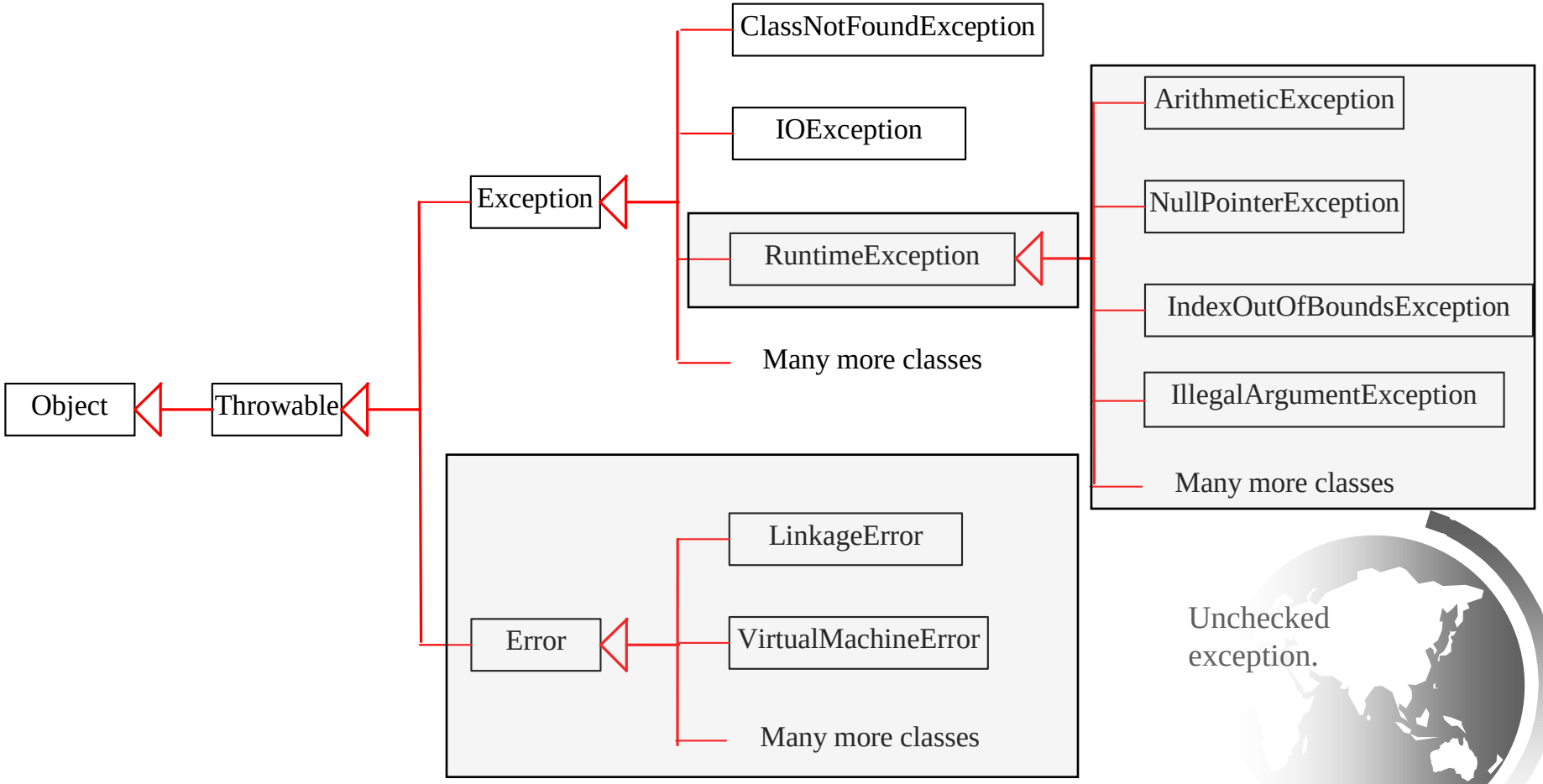
  • NullPointerException: access an object before it is assigned;

  • IndexOutOfBoundsException: access an element in an array outside the bounds of the array.

- To avoid cumbersome overuse of try-catch blocks, Java does not mandate you to catch unchecked exceptions.

* All other exceptions are *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

Unchecked Exceptions

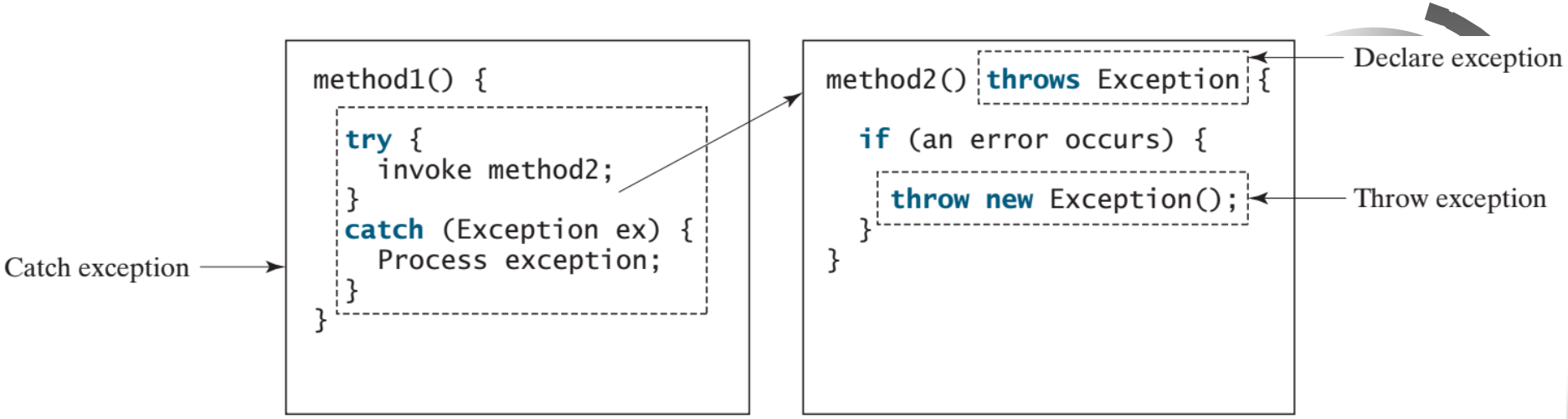


Unchecked exception.

Declaring, Throwing, and Catching Exceptions

* A handler for an exception is found by propagating the exception backward through a chain of method calls, starting from the current method.

* Java's exception-handling model is based on three operations: declaring an exception, throwing an exception, and catching an exception,



Declaring Exceptions

* *declaring exceptions*: state the types of checked exceptions it might throw.

  - only need to declare checked exceptions explicitly.

* Use the *throws* keyword in the method header to declare an exception in a method:

```
public void myMethod() throws IOException
```

* If the method might throw multiple exceptions, use:

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

Throwing Exceptions

* *throwing an exception*: when the program detects an error, it can create an instance of an appropriate exception type and throw it.

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

* When an exception is thrown, it can be caught and handled in a <u>try-catch block</u>, as follows:

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

Catching Exceptions

```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

An exception is thrown in `method3`

- If the exception type is **`Exception3`**, it is caught by the **`catch`** block for handling exception **`ex3`** in **`method2`**. **`statement5`** is skipped, and **`statement6`** is executed.
- If the exception type is **`Exception2`**, **`method2`** is aborted, the control is returned to **`method1`**, and the exception is caught by the **`catch`** block for handling exception **`ex2`** in **`method1`**. **`statement3`** is skipped, and **`statement4`** is executed.
- If the exception type is **`Exception1`**, **`method1`** is aborted, the control is returned to the **`main`** method, and the exception is caught by the **`catch`** block for handling exception **`ex1`** in the **`main`** method. **`statement1`** is skipped, and **`statement2`** is executed.
- If the exception type is not caught in **`method2`**, **`method1`**, or **`main`**, the program terminates, and **`statement1`** and **`statement2`** are not executed.

Catch or Declare Checked Exceptions

* If a method declares a checked exception, you must,
  - invoke it in a try-catch block,
  - or declare to throw the exception in the calling method.

* Example: method p1 invokes method p2, and p2 may throw a checked exception, you have to write the code in (a) or (b).

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

(a) Catch exception

```
void p1() throws IOException {

  p2();

}
```

(b) Throw exception

Getting Information from Exceptions

* An exception object contains valuable information about the exception.

* You may use the instance methods in the java.lang.Throwable class to get information regarding the exception.

java.lang.Throwable	
`+getMessage(): String`	Returns the message that describes this exception object.
`+toString(): String`	Returns the concatenation of three strings: (1) the full name of the exception class; (2) `":"` (a colon and a space); (3) the `getMessage()` method.
`+printStackTrace(): void`	Prints the `Throwable` object and its call stack trace information on the console.
`+getStackTrace(): StackTraceElement[]`	Returns an array of stack trace elements representing the stack trace pertaining to this exception object.

Rethrowing Exceptions

* Java allows an exception handler to rethrow the exception,
  - if the handler cannot process the exception
  - or simply wants to let its caller be notified of the exception.

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

The `finally` Clause

* Occasionally, you may want some code to be <u>executed regardless</u> of whether an exception occurs or is caught.

* Java has a *finally* clause that can be used to do so.

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

The `finally` Clause, cont.

* If no exception arises in the try block, final Statements is executed, and the next statement after the try statement is executed.

* If a statement causes an exception in the try block that is caught in a catch block, the rest of the statements in the try block are skipped, the catch block is executed, and the finally clause is executed. The next statement after the try statement is executed.

* If one of the statements causes an exception that is not caught in any catch block, the other statements in the try block are skipped, the finally clause is executed, and the exception is passed to the caller of this method.

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.

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;
```

Handling exception

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;
```

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

Cautions When Using Exceptions

* Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.

* Be aware, however, that 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.

When to Throw Exceptions

* If you want the exception to be processed by its caller, 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

* You should use it to deal with unexpected error conditions.

* Do not use it to deal with simple, expected situations.

Example: when to Use Exceptions

```
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");
```

Chained Exceptions

* *chained exceptions*: throw a new exception (with additional information) along with the original exception.

```
java.lang.Exception: New info from method1
  at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:16)
  at ChainedExceptionDemo.main(ChainedExceptionDemo.java:4)
Caused by: java.lang.Exception: New info from method2
  at ChainedExceptionDemo.method2(ChainedExceptionDemo.java:21)
  at ChainedExceptionDemo.method1(ChainedExceptionDemo.java:13)
  ... 1 more
```

```java
public class ChainedExceptionDemo {
  public static void main(String[] args) {
    try {
      method1();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  public static void method1() throws Exception {
    try {
      method2();
    }
    catch (Exception ex) {
       throw new Exception("New info from method1", ex);
    }
  }

  public static void method2() throws Exception {
    throw new Exception("New info from method2");
  }
}
```

Defining Custom Exception Classes

* Use the exception classes in the API whenever possible.
* Define custom exception classes if the predefined classes are not sufficient.
* By extending Exception or a subclass of Exception (java.lang.Exception).

```
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;
  }
}
```

Listing 12.10
InvalidRadiusException.java

```java
public class TestCircleWithCustomException {
  public static void main(String[] args) {
    try {
      new CircleWithCustomException(5);
      new CircleWithCustomException(-5);
      new CircleWithCustomException(0);
    }
    catch (InvalidRadiusException ex) {
      System.out.println(ex);
    }

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

```java
/** Construct a circle with a specified radius */
public CircleWithCustomException(double newRadius)
    throws InvalidRadiusException {
  setRadius(newRadius);
  numberOfObjects++;
}
```

```java
/** Set a new radius */
public void setRadius(double newRadius)
    throws InvalidRadiusException {
  if (newRadius >= 0)
    radius = newRadius;
  else
    throw new InvalidRadiusException(newRadius);
}
```

```
InvalidRadiusException: Invalid radius -5.0
Number of objects created: 1
```

The File Class

* The File class:

- provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion.

- is a wrapper class for file name and its directory path.

- contains the methods for obtaining file and directory properties and for renaming and deleting files and directories.

- does not contain the methods for reading and writing file contents.

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

Listing 12.12 TestFileClass.java

```
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()));
  }
}
```

* Constructing a File instance does not create a file on the machine.
* You can create a File instance for any file name regardless whether it exists or not.
* You can invoke the exists() method to check whether the file exists.

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.

  - There are two types of files: text and binary.

  - 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(file: File)`	Creates a `PrintWriter` object for the specified file object.
`+PrintWriter(filename: String)`	Creates a `PrintWriter` object for the specified file-name string.
`+print(s: String): void`	Writes a string to the file.
`+print(c: char): void`	Writes a character to the file.
`+print(cArray: char[]): void`	Writes an array of characters to the file.
`+print(i: int): void`	Writes an `int` value to the file.
`+print(l: long): void`	Writes a `long` value to the file.
`+print(f: float): void`	Writes a `float` value to the file.
`+print(d: double): void`	Writes a `double` value to the file.
`+print(b: boolean): void`	Writes a `boolean` value to the file.
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 §4.6, "Formatting Console Output."

```java
public class WriteData {
  public static void main(String[] args) throws IOException {
    java.io.File file = new java.io.File("scores.txt");
    if (file.exists()) {
      System.out.println("File already exists");
      System.exit(1);
    }

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

    // 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();
  }
}
```

```
John T Smith 90
Eric K Jones 85
```

scores.txt

Reading Data Using Scanner

* A Scanner breaks its input into tokens delimited by whitespace characters.

* To read from the keyboard, you create a Scanner for System.in, as follows:

```
Scanner input = new Scanner(System.in);
```

* To read from a file, create a Scanner for a file, as follows:

```
Scanner input = new Scanner(new File(filename));
```

Reading Data Using Scanner , cont.

java.util.Scanner	
+Scanner(source: File)	Creates a Scanner that scans tokens from the specified file.
+Scanner(source: String)	Creates a Scanner that scans tokens from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has more data to be read.
+next(): String	Returns next token as a string from this scanner.
+nextLine(): String	Returns a line ending with the line separator from this scanner.
+nextByte(): byte	Returns next token as a byte from this scanner.
+nextShort(): short	Returns next token as a short from this scanner.
+nextInt(): int	Returns next token as an int from this scanner.
+nextLong(): long	Returns next token as a long from this scanner.
+nextFloat(): float	Returns next token as a float from this scanner.
+nextDouble(): double	Returns next token as a double from this scanner.
+useDelimiter(pattern: String): Scanner	Sets this scanner's delimiting pattern and returns this scanner.

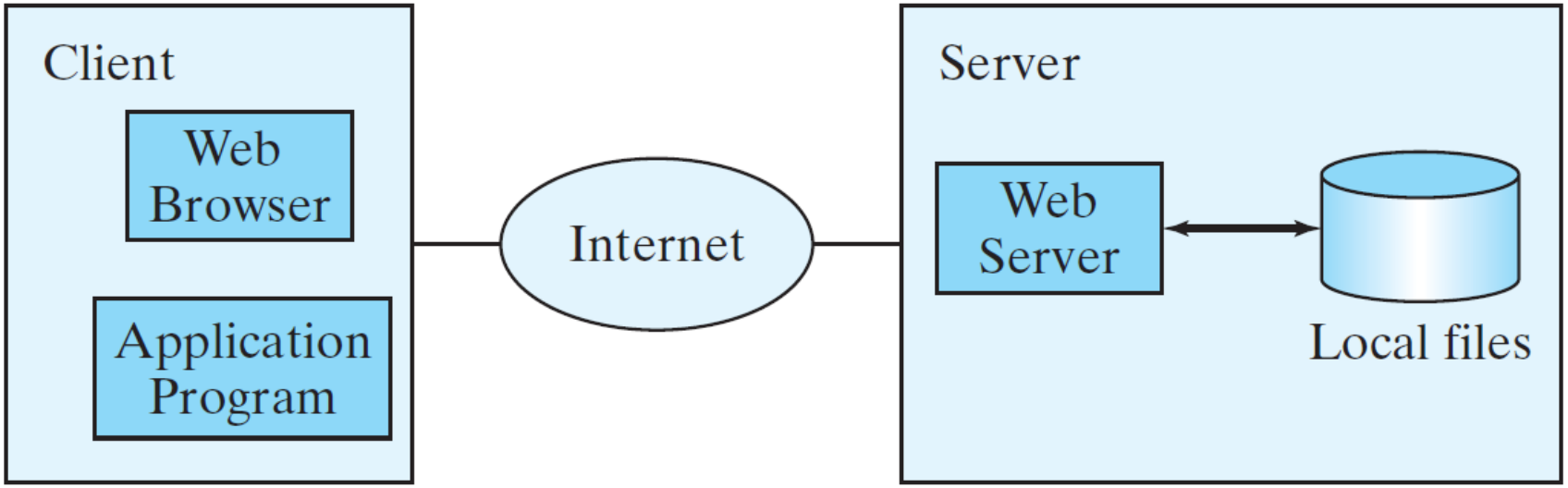
```java
import java.util.Scanner;

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 firstName = input.next();
    String mi = input.next();
    String lastName = input.next();
    int score = input.nextInt();
    System.out.println(
      firstName + " " + mi + " " + lastName + " " + score);
  }

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

scores.txt

```
John T Smith 90
Eric K Jones 85
```

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());

Listing 12.17 ReadFileFromURL.java

```java
import java.util.Scanner;

public class ReadFileFromURL {
  public static void main(String[] args) {
    System.out.print("Enter a URL: ");
    String URLString = new Scanner(System.in).next();

    try {
      java.net.URL url = new java.net.URL(URLString);
      int count = 0;
      Scanner input = new Scanner(url.openStream());
      while (input.hasNext()) {
        String line = input.nextLine();
        count += line.length();
      }

      System.out.println("The file size is " + count + " characters");
    }
    catch (java.net.MalformedURLException ex) {   // URL isn't formed correctly
      System.out.println("Invalid URL");
    }
    catch (java.io.IOException ex) {              // URL does not exist
      System.out.println("I/O Errors: no such file");
    }
  }
}
```

```
Enter a URL: http://cs.armstrong.edu/liang/data/Lincoln.txt ↵ Enter
The file size is 1469 characters
```