13 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 Overview

Show runtime error

Quotient

Run

Fix it using an if statement

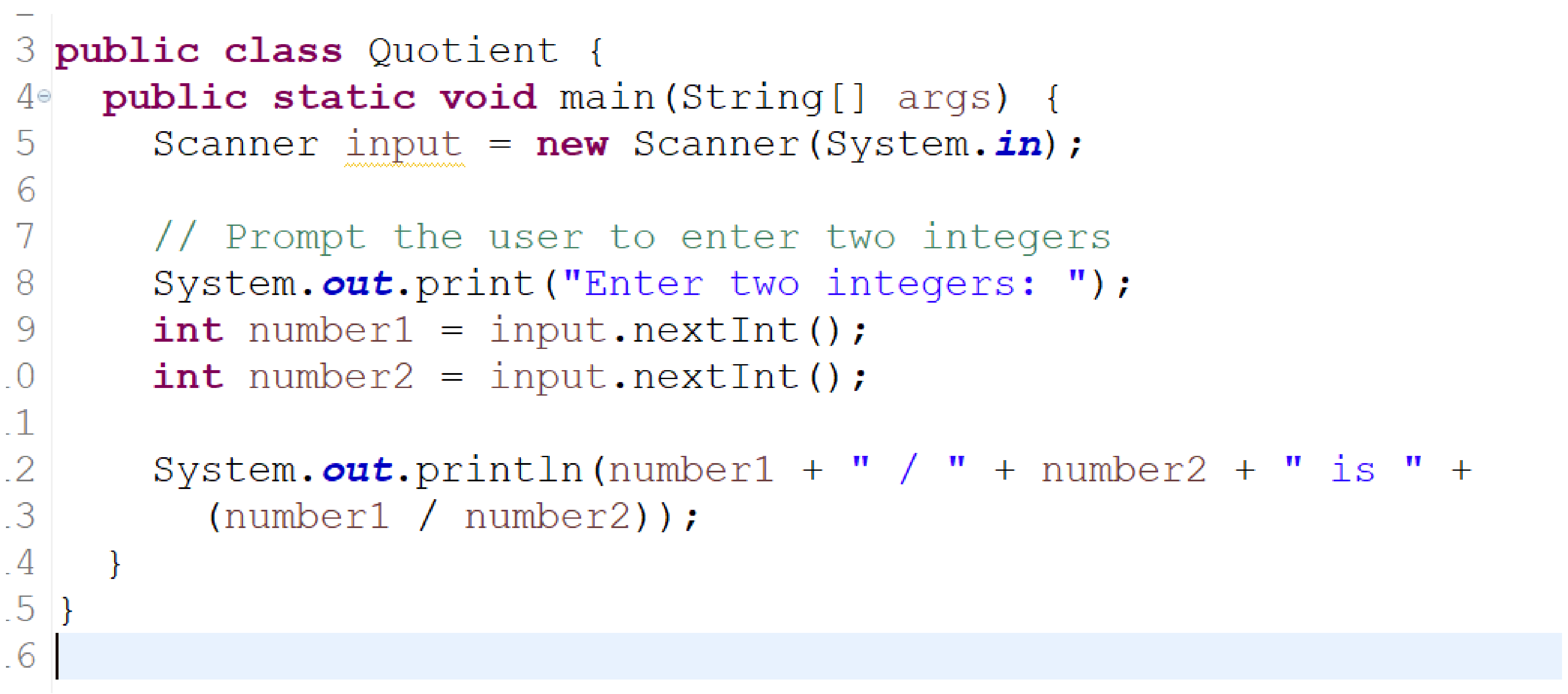
QuotientWithIf

Run

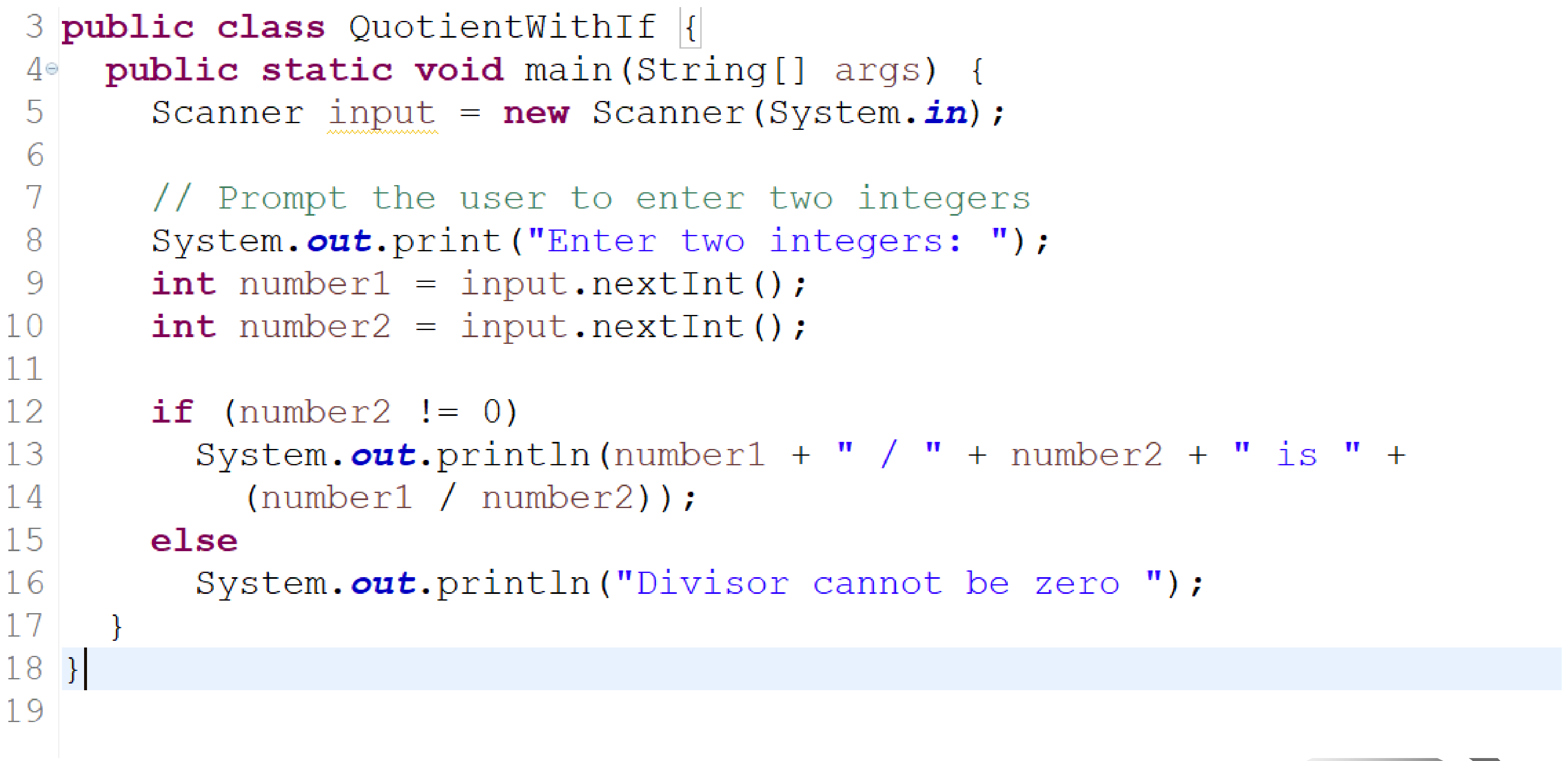
With a method

QuotientWithMethod

Run



```java
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));
  }
}

```



```java
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 ");
  }
}
```

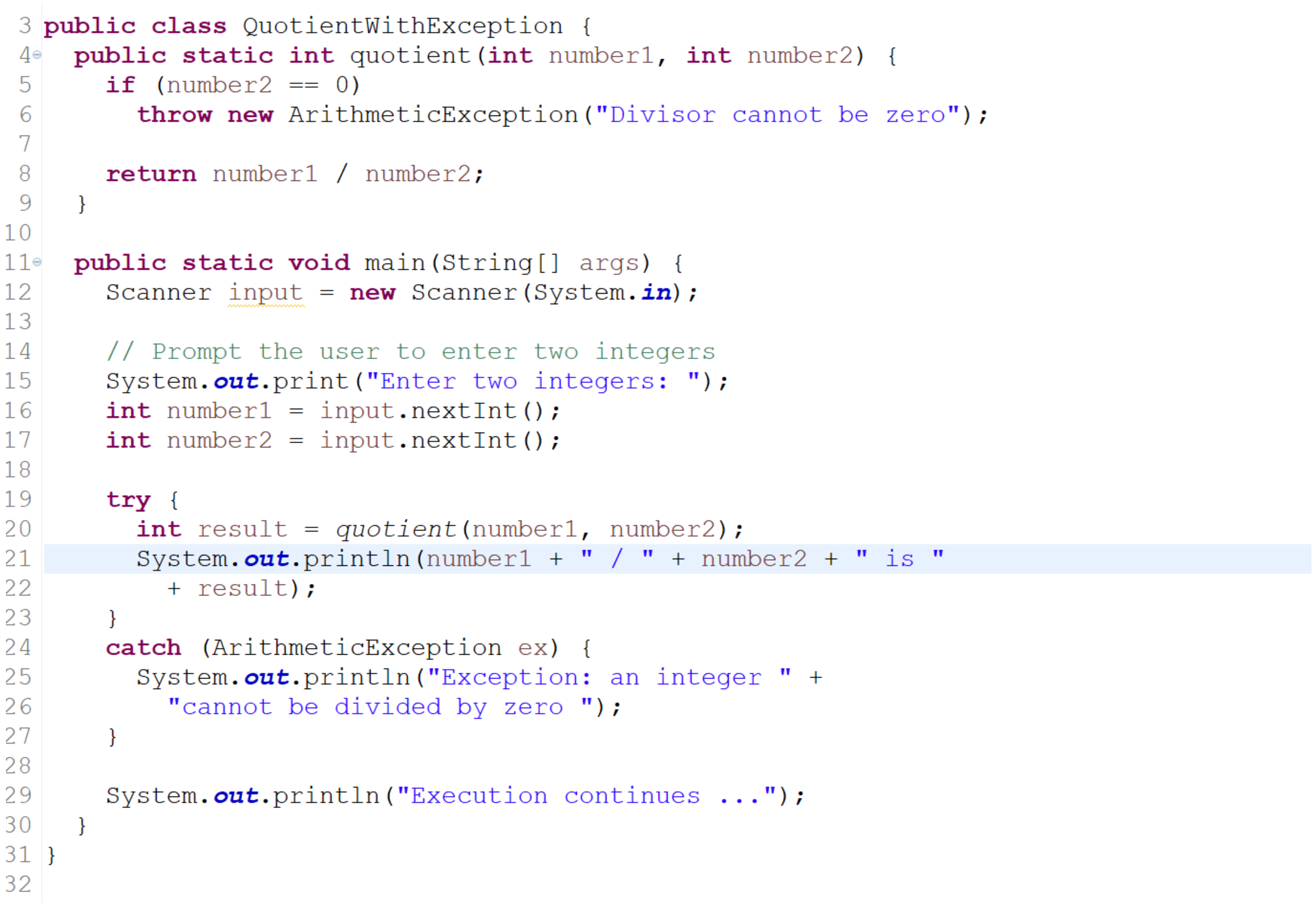
```java
public class QuotientWithMethod {
  public static int quotient(int number1, int number2) {
    if (number2 == 0) {
      System.out.println("Divisor cannot be zero");
      System.exit(1);
    }

    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);
  }
}
```

Exception Advantages

QuotientWithException

Run

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.



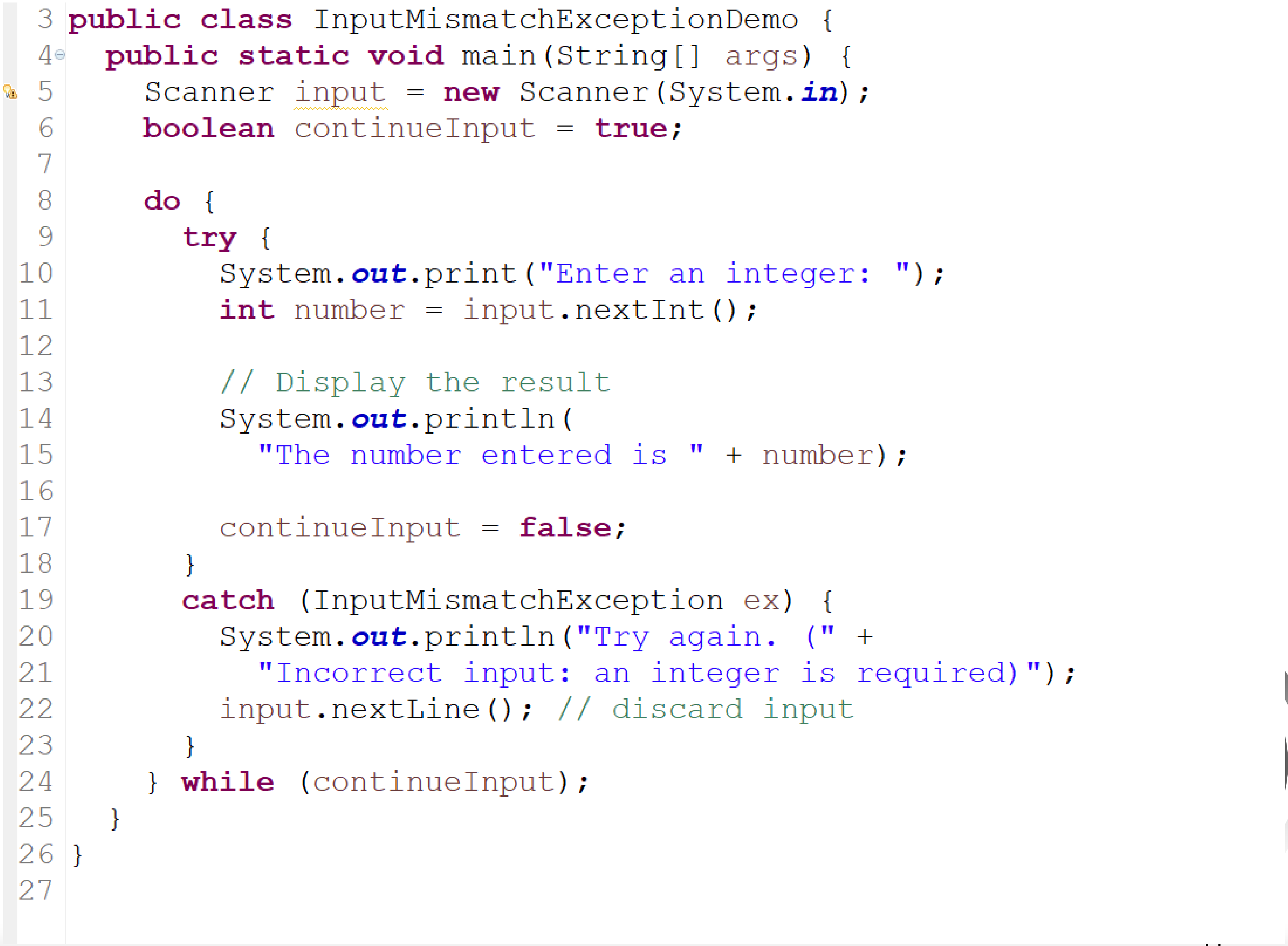
```java
public class QuotientWithException {
  public static int quotient(int number1, int number2) {
    if (number2 == 0)
      throw new ArithmeticException("Divisor cannot be zero");

    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 ...");
  }
}

```

Handling InputMismatchException

InputMismatchExceptionDemo

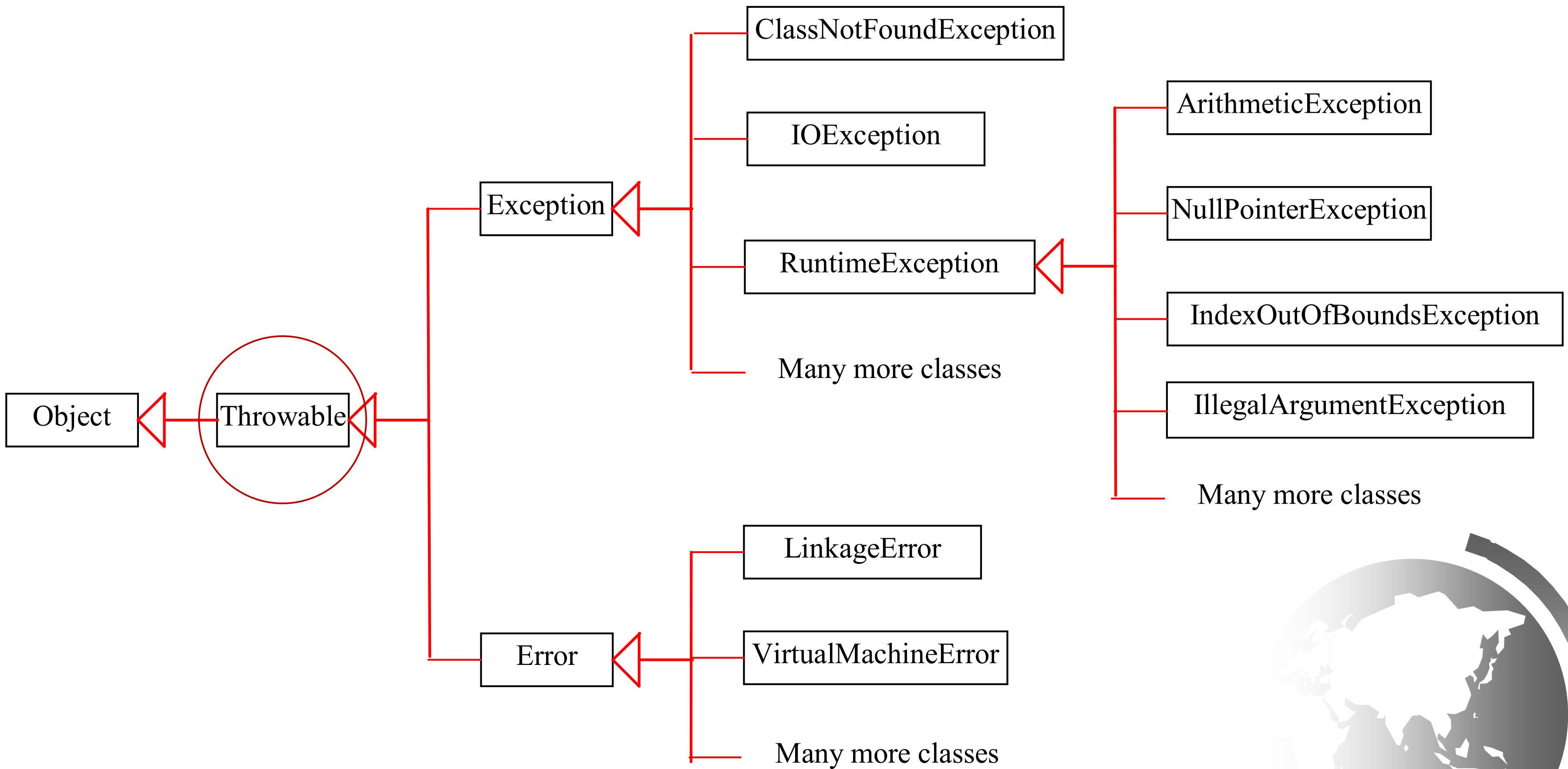
Run

By handling InputMismatchException, your program will continuously read an input until it is correct.

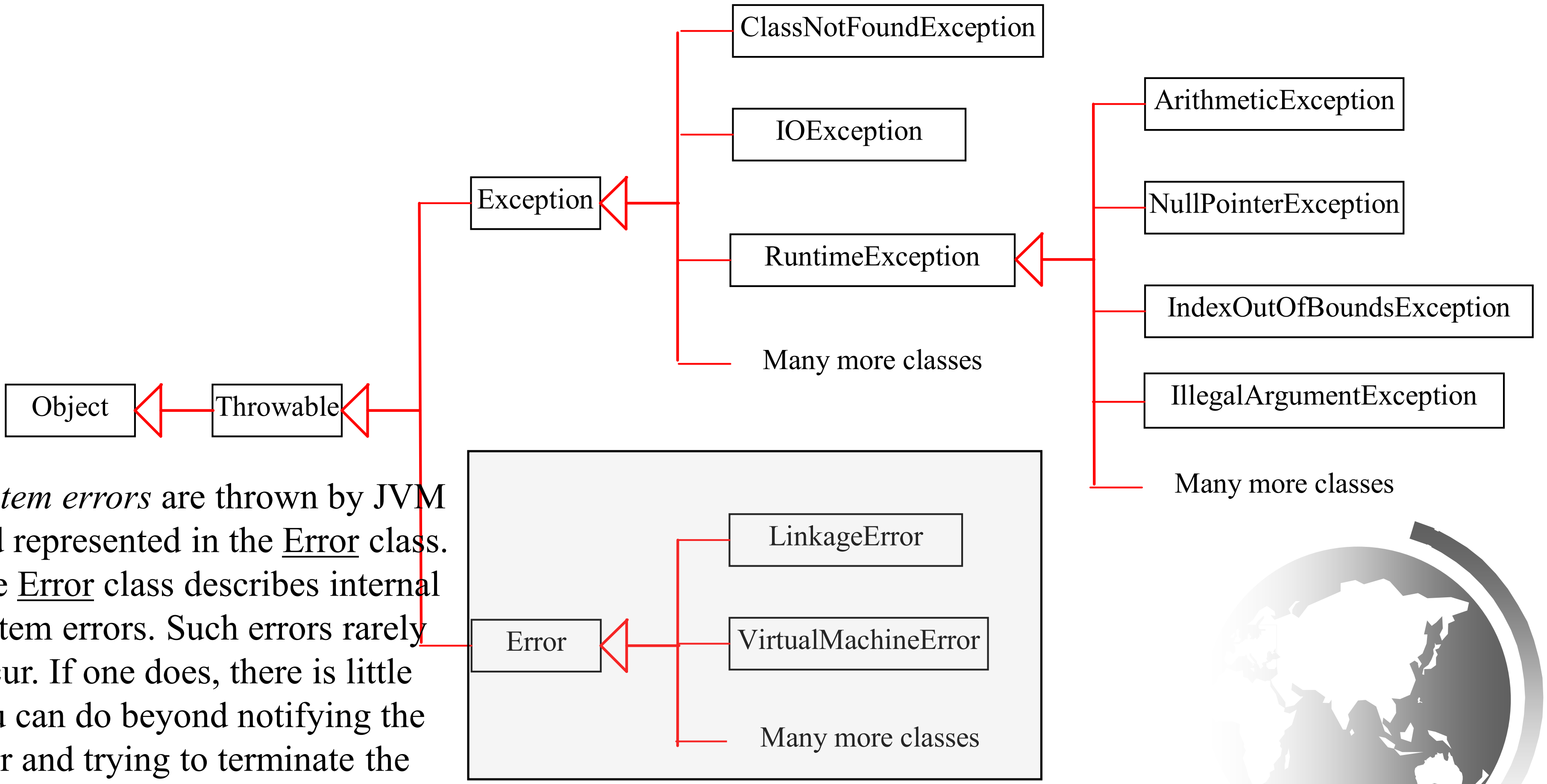


```java
public class InputMismatchExceptionDemo {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    boolean continueInput = true;

    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 Types



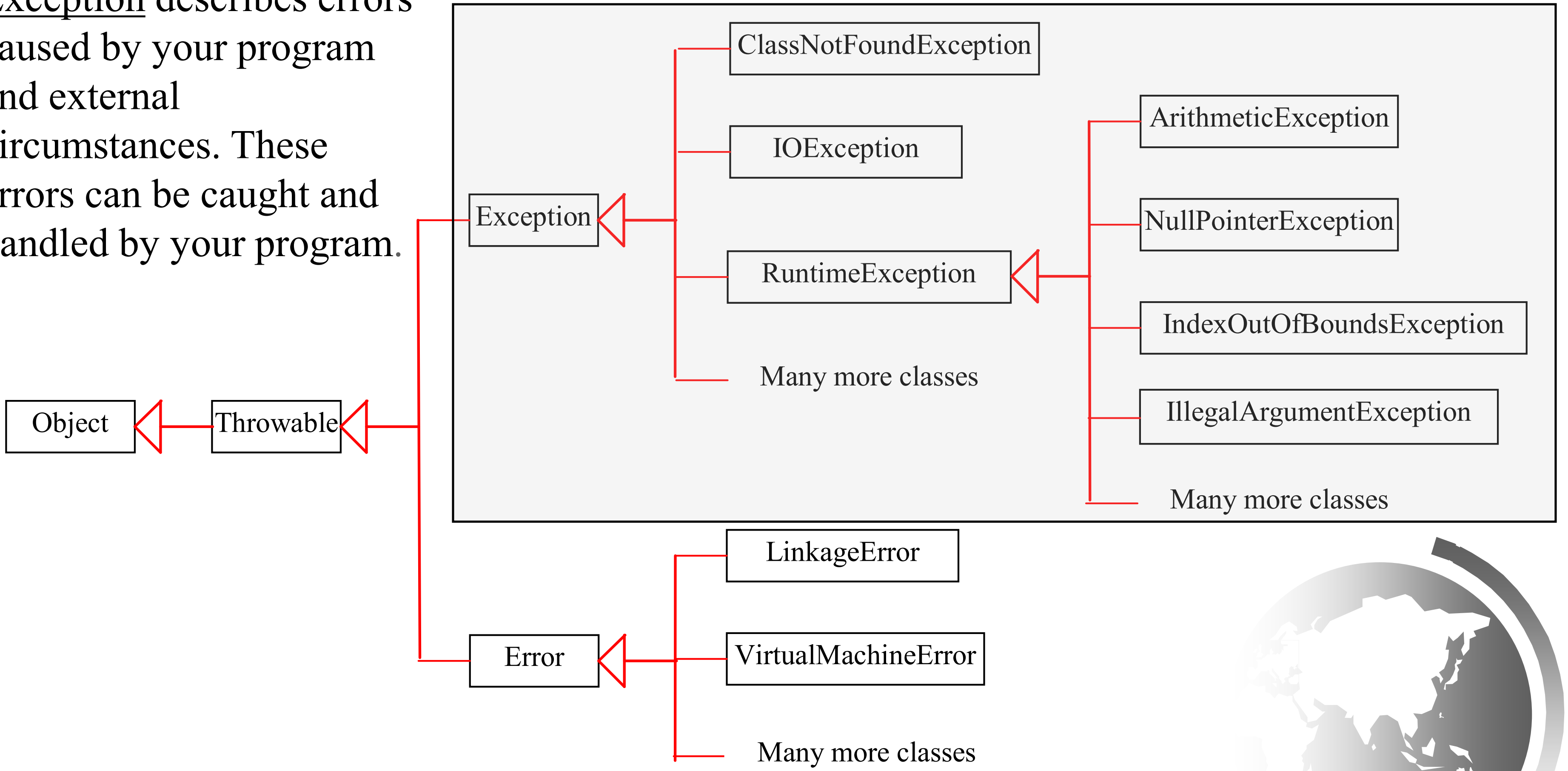
System Errors



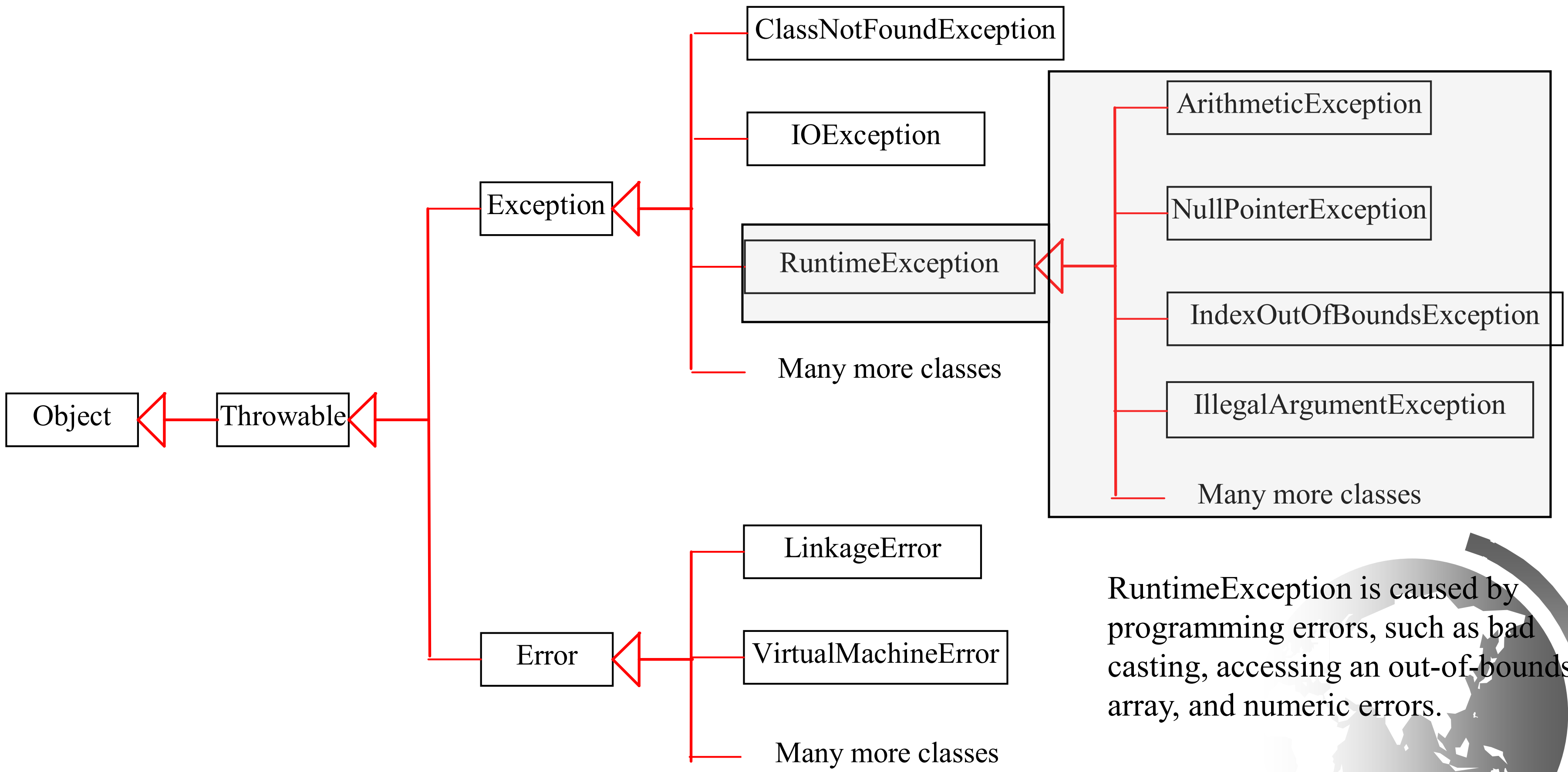
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

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.

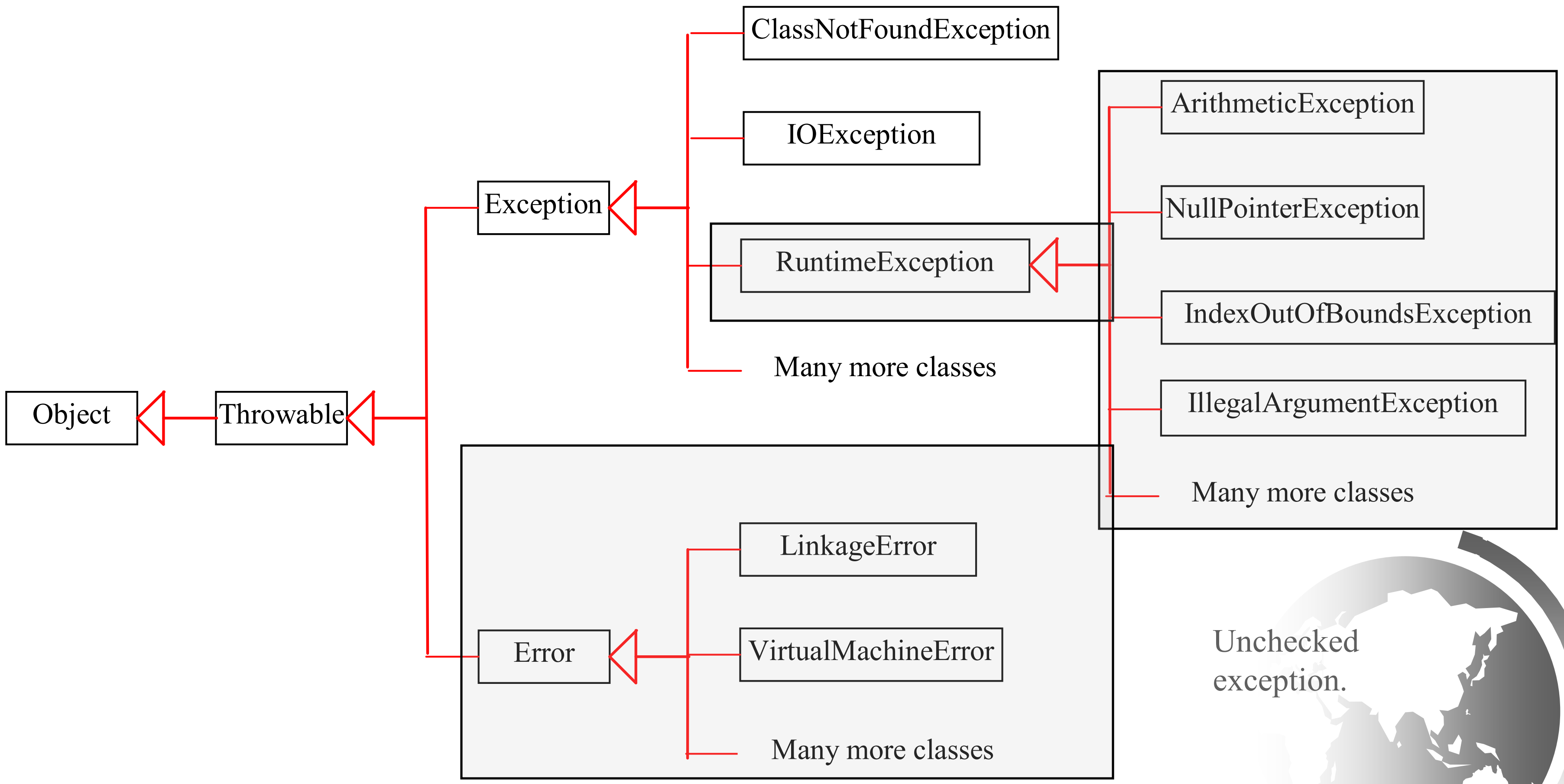
Unchecked Exceptions

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.

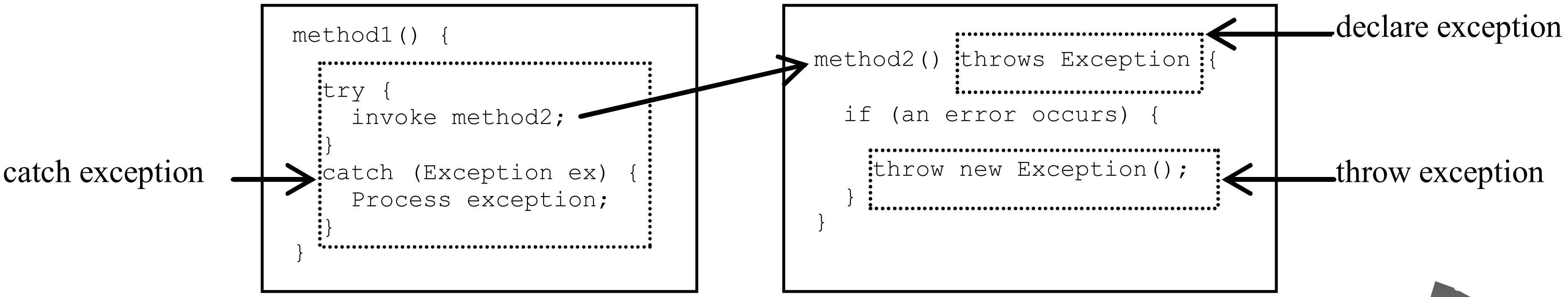
Unchecked exceptions can occur anywhere in the program. To avoid cumbersome overuse of try-catch blocks, Java does not mandate you to write code to catch unchecked exceptions.

Unchecked Exceptions



Unchecked exception.

Declaring, Throwing, and Catching Exceptions



```
method1() {

  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

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

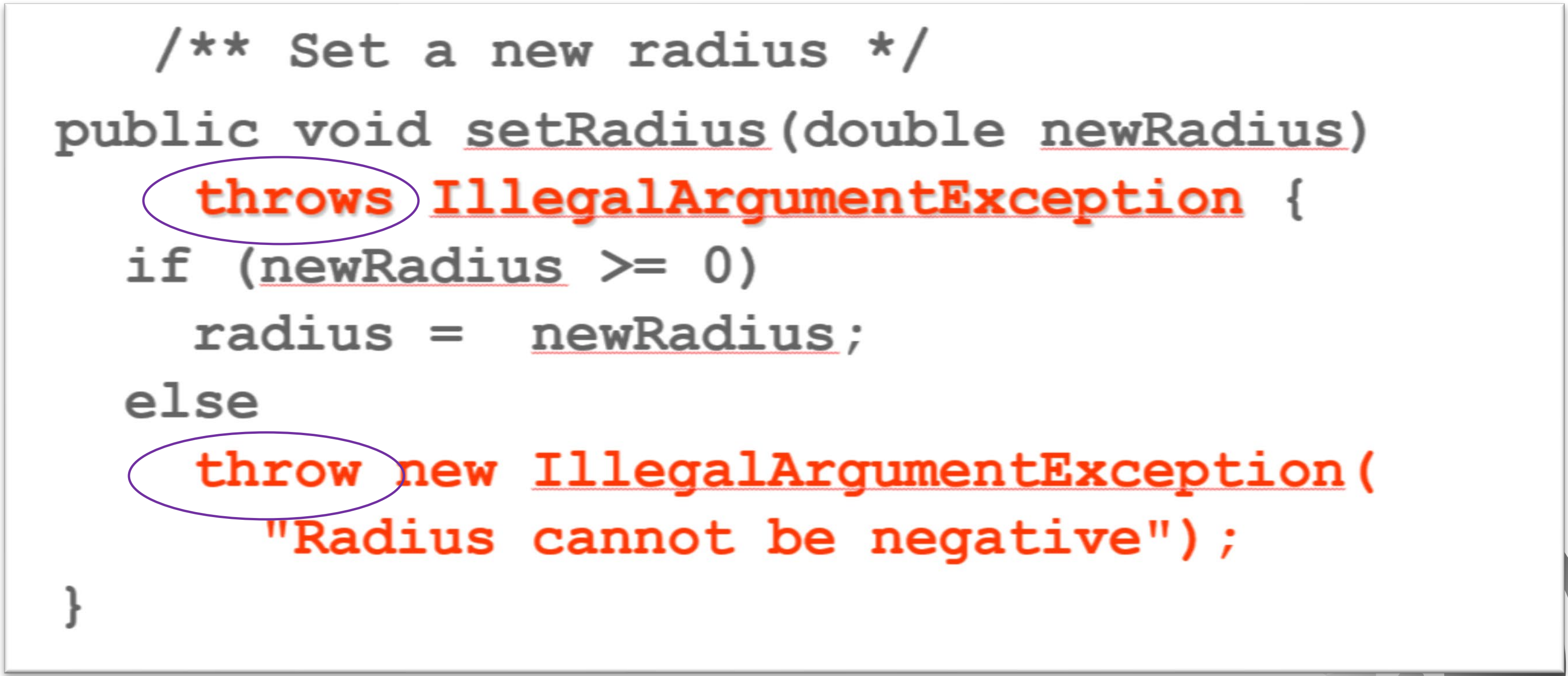
When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example,

```
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");
}
```

throw VS. throws

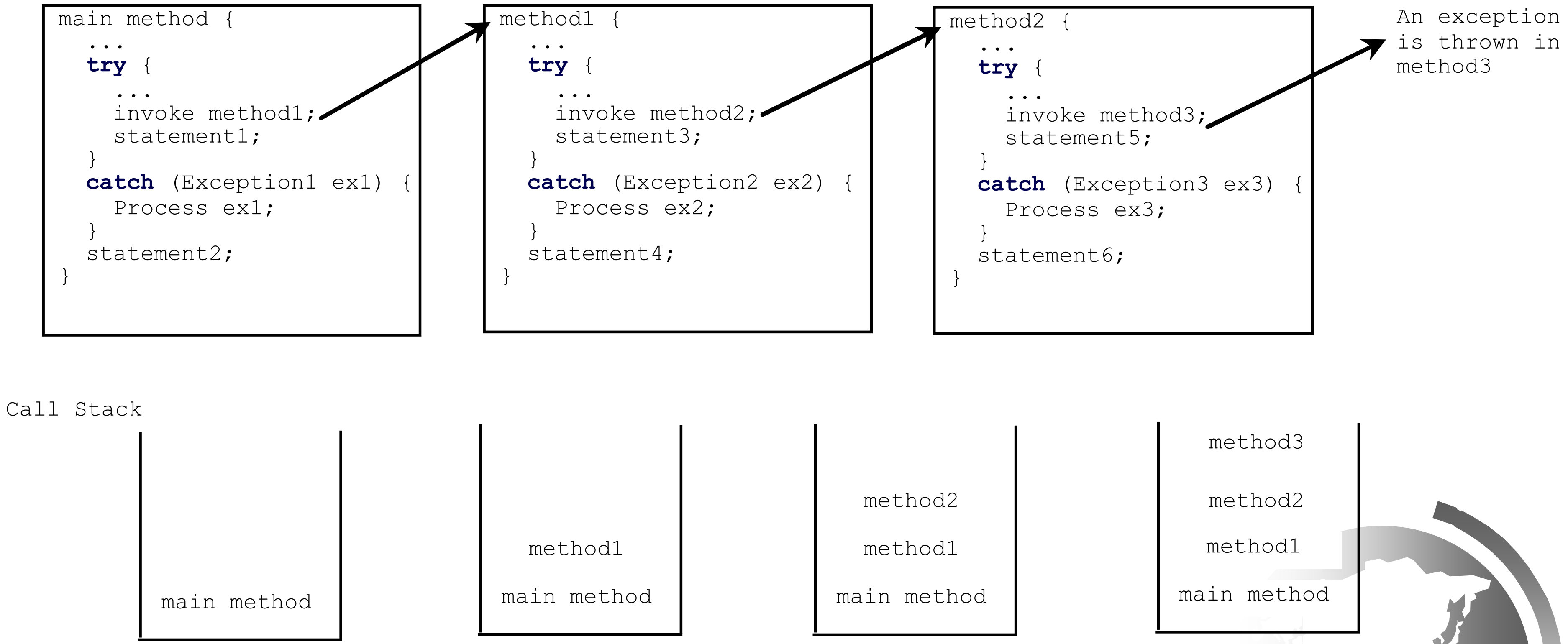


```
   /** Set a new radius */
public void setRadius(double newRadius)
    throws IllegalArgumentException {
  if (newRadius >= 0)
    radius =  newRadius;
  else
    throw new IllegalArgumentException(
      "Radius cannot be negative");
}
```

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



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

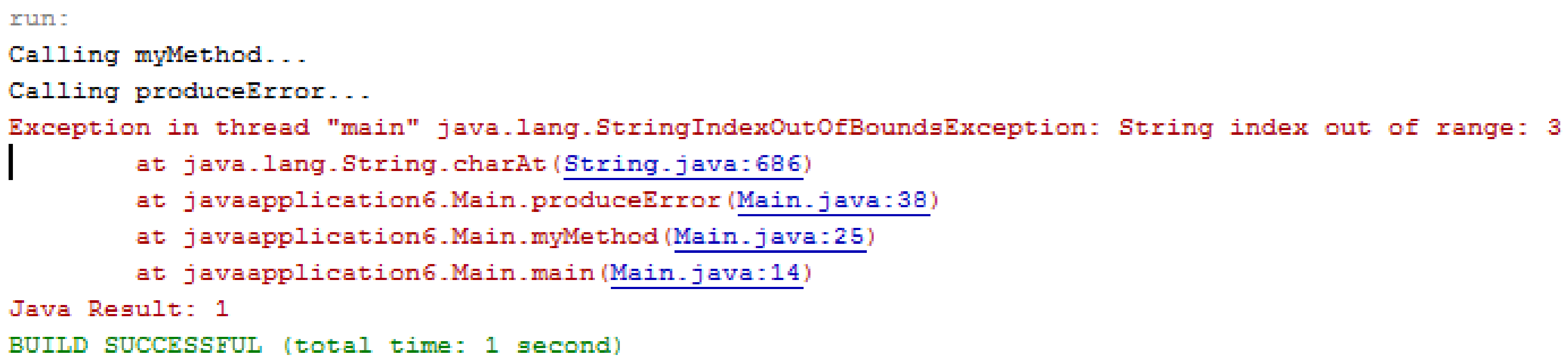
```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

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

The Stack Trace

- The *call stack* is an internal list of all the methods that are currently executing.
- A *stack trace* is a list of all the methods in the call stack.
- It indicates:
  - the method that was executing when an exception occurred and
  - all of the methods that were called in order to execute that method.
- Example: StackTrace.java

```
public class StackTrace
{
  public static void main(String[] args)
  {
    System.out.println("Calling myMethod...");
    myMethod();
    System.out.println("Method main is done.");
  }

  public static void myMethod()
  {
    System.out.println("Calling produceError...");
    produceError();
    System.out.println("myMethod is done.");
  }

  public static void produceError()
  {
     String str = "abc";

     System.out.println(str.charAt(3));
     System.out.println("produceError is done.");
  }
}
```



```
run:
Calling myMethod...
Calling produceError...
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 3
        at java.lang.String.charAt(String.java:686)
        at javaapplication6.Main.produceError(Main.java:38)
        at javaapplication6.Main.myMethod(Main.java:25)
        at javaapplication6.Main.main(Main.java:14)
Java Result: 1
BUILD SUCCESSFUL (total time: 1 second)
```

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)

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.

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

这条语句不会执行，因为已经throw异常了

```
class Test {
  public static void main(String[] args) {
    try {
      System.out.println("Welcome to Java");
      int i = 0;
      int y = 2 / i;
      System.out.println("Welcome to HTML");
    }
    finally {
      System.out.println("The finally clause is executed");
    }
    System.out.println("End of the block");
}}
```

NO catch

- 输出：

Welcome to Java
The finally clause is executed

Java Exception使用规范

- **1. 原则：不要忽略checked Exception**

```
try
{
    method1();  //method1抛出ExceptionA
}
catch(ExceptionA e)
{
    e.printStackTrace();
}
```

这段代码捕获了异常却不作任何处理，可以算得上Java编程中的杀手。调用一下printStackTrace算不上“处理异常”。调用printStackTrace对调试程序有帮助，但程序调试阶段结束之后，printStackTrace就不应再在异常处理模块中担负主要责任了。

Java Exception使用规范

- 应该如何处理呢，这里有四个选择：
  - （1）**处理异常**，进行修复以让程序继续执行。例如在进行数据库查询时，数据库连接断链后重建链接成功。
  - （2）在对异常进行分析后发现这里不能处理它，那么**重新抛出异常**，让调用者处理。异常依次向上抛出，如果所有方法都不能恰当地处理异常，最终会在用户界面以恰当的方式提示用户，由用户来判断下一步处理措施。例如在进行数据库查询时，断链后重试几次依然失败的情况。
  - （3）将异常**转换为其他异常再抛出**，这时应该注意不要丢失原始异常信息。这种情况一般用于将底层异常封装为应用层异常。
  - （4）**不捕获异常**,直接在函数定义中使用throws声明将抛出该异常。让调用者去处理该异常。
  - 因此，当捕获一个checked Exception的时候，必须对异常进行处理；如果认为不必要在这里作处理，就不要捕获该异常，在方法体中声明方法抛出异常，由上层调用者来处理该异常。

Java Exception使用规范

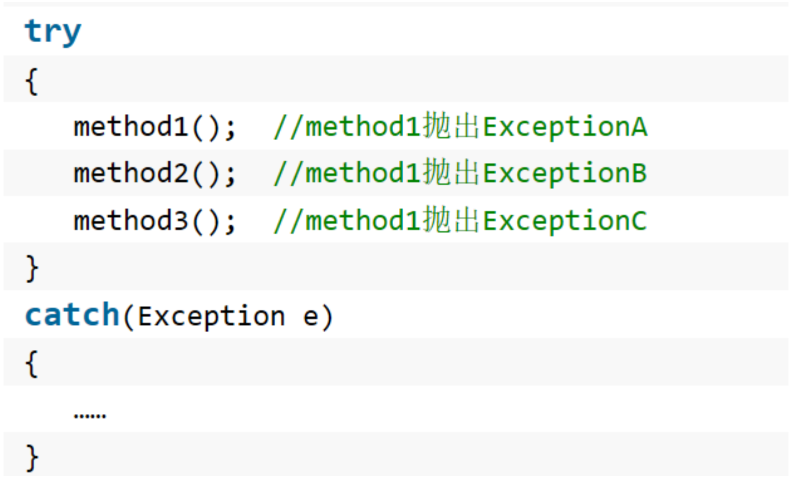
- **2. 建议：不要捕获unchecked Exception**
  - 有两种unchecked Exception：
  - Error：这种情况属于JVM发生了不可恢复的故障，例如内存溢出，无法处理。
  - RuntimeException：这种情况属于错误的编码导致的，出现异常后需要修改代码才能修复，一般来说catch后没有恰当的处理方式，因此不应该捕获。

例如由于编码错误，下面的代码会产生ArrayIndexOutofBoundException。修改代码后才能修复该异常。

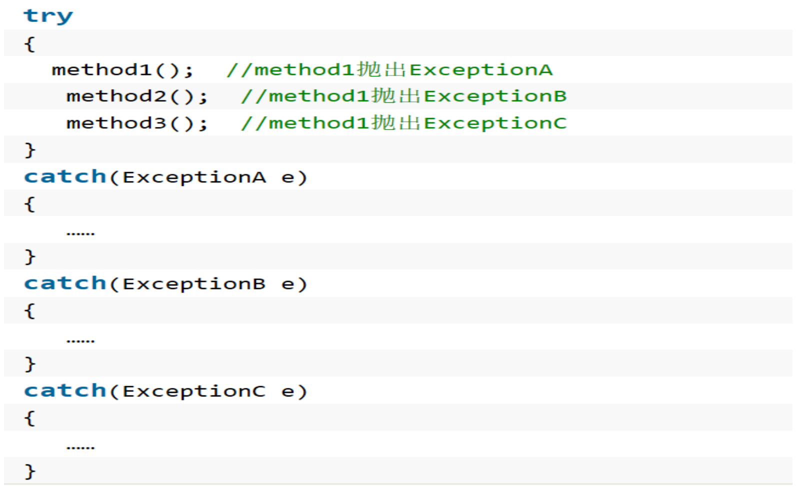
```
int[] intArray = new int[10];
intArray[10]=1;
```

Java Exception使用规范

- **3. 原则：不要一次捕获所有的异常**



```
try
{
    method1();  //method1抛出ExceptionA
    method2();  //method1抛出ExceptionB
    method3();  //method1抛出ExceptionC
}
catch(Exception e)
{
    ……
}
```



```
try
{
  method1();  //method1抛出ExceptionA
   method2();  //method1抛出ExceptionB
   method3();  //method1抛出ExceptionC
}
catch(ExceptionA e)
{
   ……
}
catch(ExceptionB e)
{
   ……
}
catch(ExceptionC e)
{
   ……
}
```

这里有两个潜在的缺陷:

一是针对try块中抛出的每种Exception，很可能需要不同的处理和恢复措施，而由于这里只有一个catch块，分别处理就不能实现。

二是try块中还可能抛出RuntimeException,代码中捕获了所有可能抛出的RuntimeException而没有作任何处理，掩盖了编程的错误，会导致程序难以调试。

Java Exception使用规范

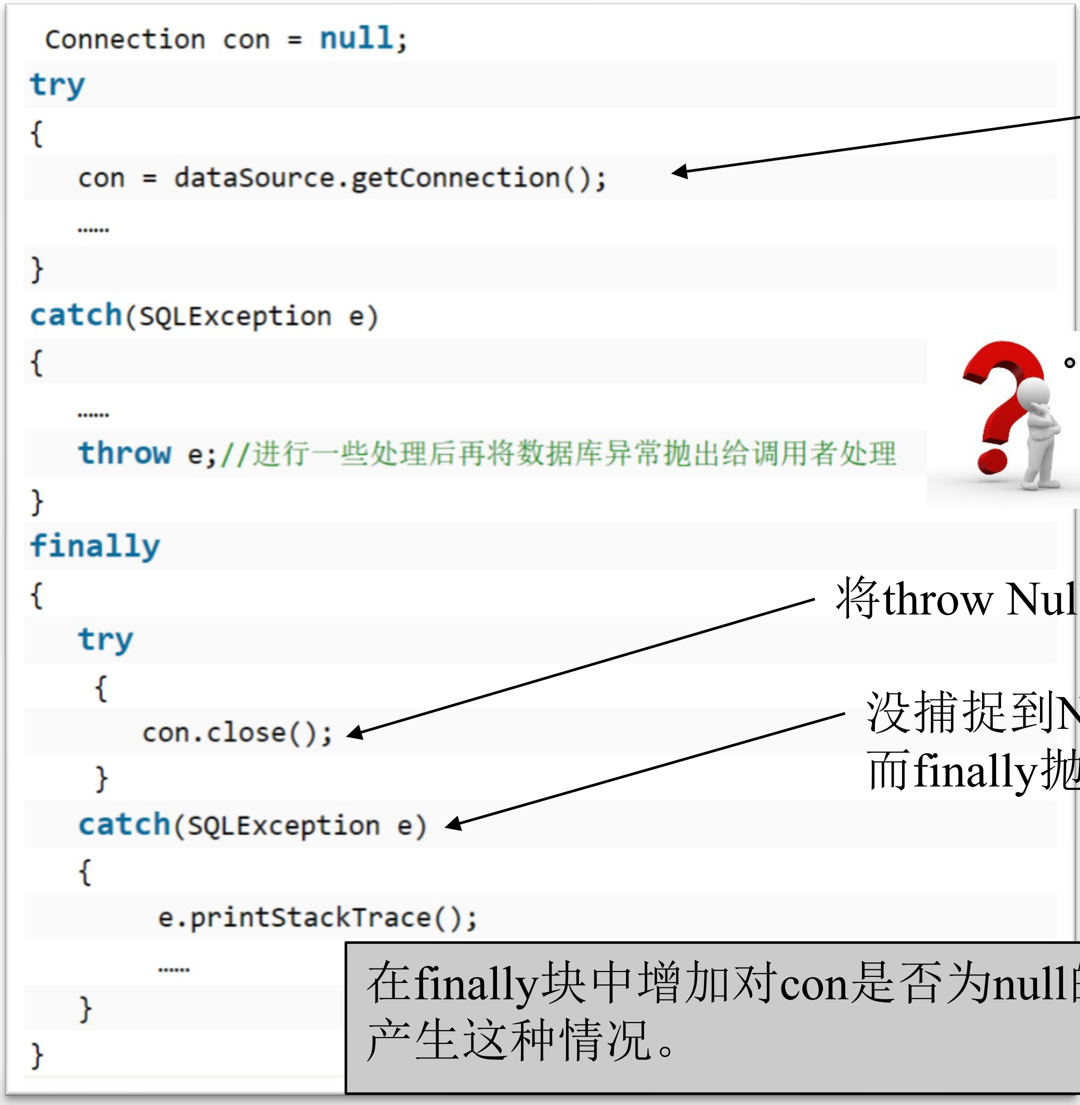
- **4 原则：使用finally块释放资源**
  - 资源：程序中使用的数量有限的对象，或者只能独占式访问的对象。例如：文件、线程、线程池、数据库连接、ftp连接。因为资源是“有限的”，因此资源使用后必须释放，以避免程序中的资源被耗尽，影响程序运行。某些资源，使用完毕后会自动释放，如线程。某些资源则需要显示释放，如数据库连接。

finally关键字保证无论程序使用任何方式离开try块，finally中的语句都会被执行。在以下情况下，finally块的代码都会执行：
（1）try块中的代码正常执行完毕。
（2）在try块中抛出异常。
（3）在try块中执行return、break、continue。
（4） catch块中代码执行完毕。
（5）在catch块中抛出异常。
（6）在catch块中执行return、break、continue。

Java Exception使用规范

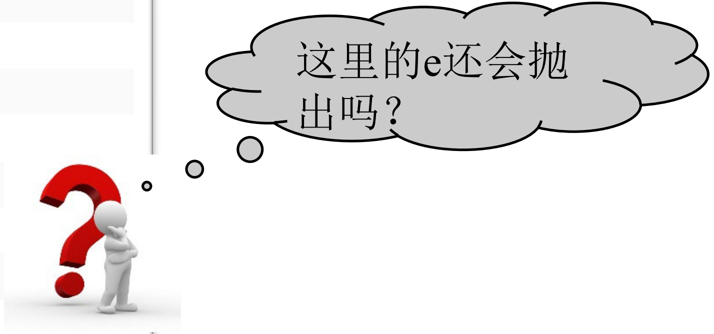
- **5. 原则：finally块不能抛出异常**
  - JAVA异常处理机制保证无论在任何情况下都先执行finally块的代码，然后再离开整个try，catch，finally块。
  - 在try，catch块中向外抛出异常的时候，JAVA虚拟机先转到finally块执行finally块中的代码，finally块执行完毕后，再将异常抛出。
  - 但如果在finally块中抛出异常，try，catch块的异常就不能抛出，外部捕捉到的异常就是finally块中的异常信息，而try，catch块中发生的真正的异常堆栈信息则丢失了。

Java Exception使用规范



```
 Connection con = null;
try
{
   con = dataSource.getConnection();
   ……
}
catch(SQLException e)
{
   ……
   throw e;//进行一些处理后再将数据库异常抛出给调用者处理
}
finally
{
   try
    {
       con.close();
    }
   catch(SQLException e)
   {
       e.printStackTrace();
       ……
   }
}
```

没有得到connection，con仍是null



将throw NullPointerException

没捕捉到NullPointerException，从而finally抛出NullPointerException

在finally块中增加对con是否为null的判断可以避免产生这种情况。

Java Exception使用规范

- **6. 原则：抛出自定义异常异常时带上原始异常信息**

```
public void method2()
{
    try
    {
        ……
        method1();  //method1进行了数据库操作
    }
    catch(SQLExceptione)
   {
        ……
        throw new MyException(“发生了数据库异常:”+e.getMessage);
    }
}
public void method3()
{
    try
    {
        method2();
    }
    catch(MyExceptione)
    {
        e.printStackTrace();
         ……
    }
}
```

原始异常SQLException的信息丢失了，这里只能看到method2里面定义的MyException的堆栈情况；而method1中发生的数据库异常的堆栈则看不到

在JDK1.4.1中，Throwable类增加了两个构造方法, public Throwable(Throwable cause)和public Throwable(String message, Throwable cause)，在构造函数中传入的原始异常堆栈信息将会在printStackTrace方法中打印出来。

Java Exception使用规范

- 7. **原则：打印异常信息时带上异常堆栈**

```java
public void method3()
{
    try
    {
        method2();
    }
    catch(MyExceptione)
   {
            ……//对异常进行处理
            System.out.println(e);//打印异常信息
   }
}
```

System.out.println(e)相当于System.out.println(e.toString()),不能打印异常堆栈，不利于事后对异常进行分析。

正确的打印方式：

```java
public void method3()
{
    try
    {
        method2();
    }
    catch(MyExceptione)
    {
            ……//对异常进行处理
            e.printStackTrace();//打印异常信息
    }
}
```

Java Exception使用规范

- **8. 原则：不要使用“异常机制+返回值” 来进行异常处理**

```
try
{
    doSomething();
}
catch(MyException e)
{
    if(e.getErrcode == -1)
    {
        ……
    }
    if(e.getErrcode == -2)
    {
        ……
    }
    ……
}
```

```
try
{
    doSomething();  //抛出MyExceptionA和MyExceptionB
}
catch(MyExceptionA e)
{
    ……
}
catch(MyExceptionB e)
{
    ……
}
```

如果有多种不同的异常情况，就定义多种不同的异常，而不要像上面代码那样综合使用Exception和返回值。

Java Exception使用规范

- **9. 建议：不要让try块过于庞大**
  - 出于省事的目的，很多人习惯于用一个庞大的try块包含所有可能产生异常的代码，
  - 这样有两个坏处：
    - 阅读代码的时候，在try块冗长的代码中，不容易知道到底是哪些代码会抛出哪些异常，不利于代码维护。
    - 使用try捕获异常是以程序执行效率为代价的，将不需要捕获异常的代码包含在try块中，影响了代码执行的效率。

Java Exception使用规范

- **10. 原则：守护线程中需要catch runtime exception**
  – 守护线程是指在需要长时间运行的线程，其生命周期一般和整个程序的时间一样长，用于提供某种持续的服务。例如服务器的告警定时同步线程，客户端的告警分发线程。
  – 由于守护线程需要长时间提供服务，因此需要对runtime exception进行保护，避免因为某一次偶发的异常而导致线程被终止。

```
while (true)
{
    try
    {
        doSomethingRepeted();
    }
    catch(MyExceptionA e)
    {
       //对checkedexception进行恰当的处理
       ……
    }
    catch(RuntimeException e)
    {
        //打印运行期异常，用于分析并修改代码
        e.printStackTrace();
    }
}
```

Java Exception使用规范

- try只与catch语句块使用时，可以使用多个catch语句来捕获try语句块中可能发生的多种异常。
- 异常发生后，Java虚拟机会**由上而下来检测**当前catch语句块所捕获的异常是否与try语句块中某个发生的异常匹配，**若匹配，则不执行其他的catch语句块**。
- 如果多个catch语句块捕获的是同种类型的异常，则**捕获子类异常的catch语句块要放在捕获父类异常的catch语句块前面**。

```
public class X {
    public static void main(String [] args) {
        try {
            badMethod();
            System.out.print("A");
        } catch (RuntimeException ex) {
            System.out.print("B");
        } catch (Exception ex1) {
            System.out.print("C");
        } finally {
            System.out.print("D");
        }
        System.out.print("E");
    }
    public static void badMethod() {
        throw new RuntimeException();
}}
```

- 输出：   BDE

```
try{
        throw new B();
} catch(A a){ System.out.println("Exception A");
} catch(B b){ System.out.println("Exception B");
}
class A extends Exception
{}
class B extends A
{}
```

Exception in thread "main" java.lang.Error: Unresolved compilation problem: Unreachable catch block for B. It is already handled by the catch block for A.

It prints:(2分)

A. Exception A
B. Exception B
C. Compile error
D. Compiled but exception raises at run-time

答案是C，子类异常应该放父类异常上面，写法有问题

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

- An exception occurs in a method.
- 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

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");
```

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

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.**

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

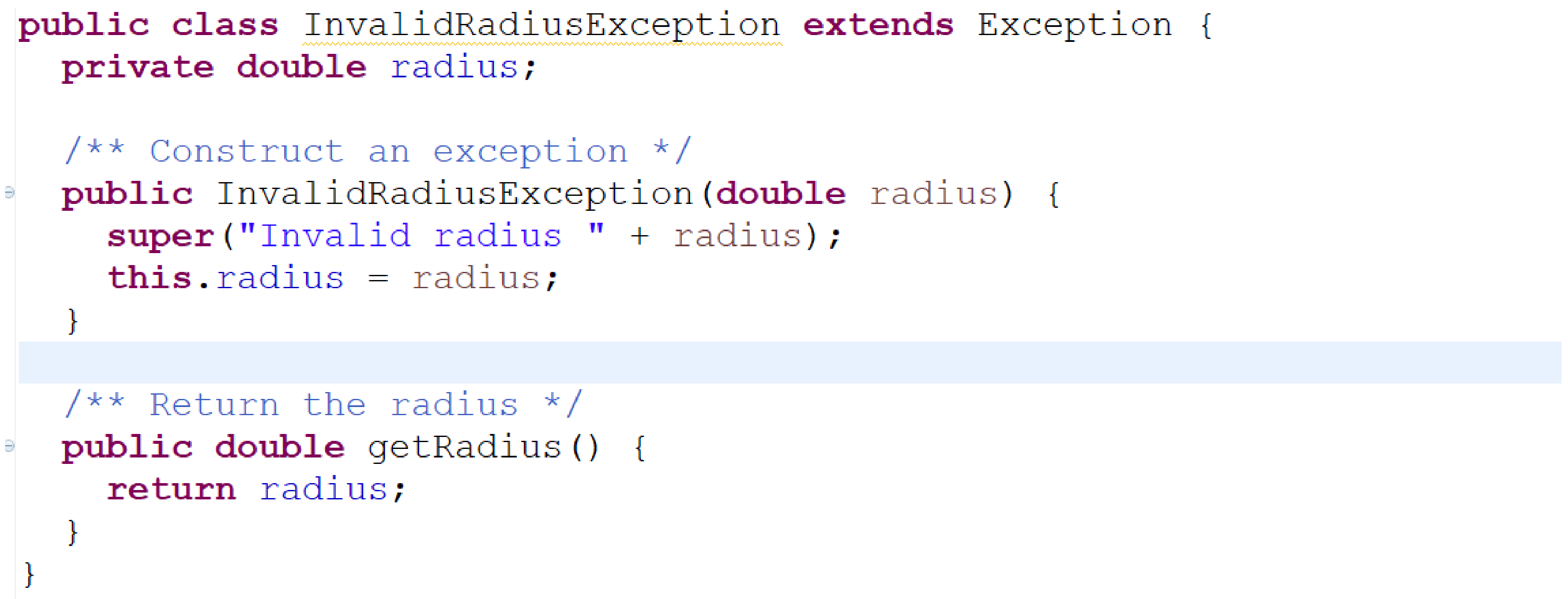
Custom Exception Class Example

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.**

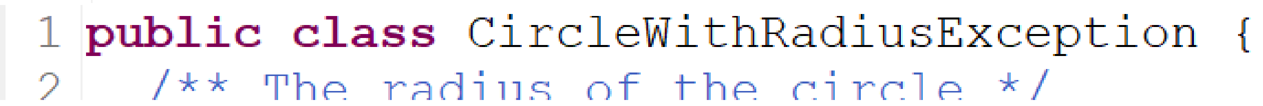
InvalidRadiusException

CircleWithRadiusException

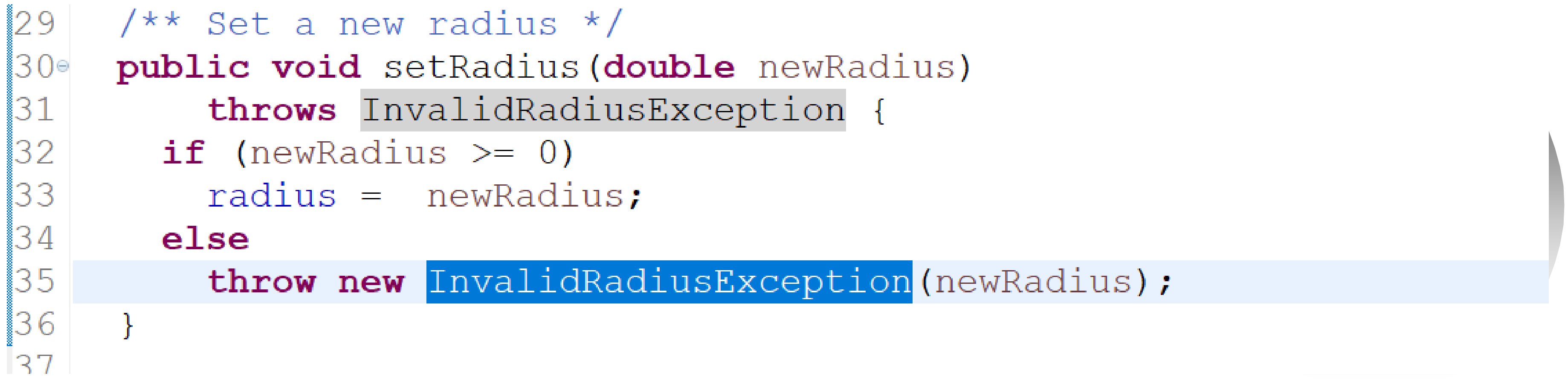
TestCircleWithRadiusException



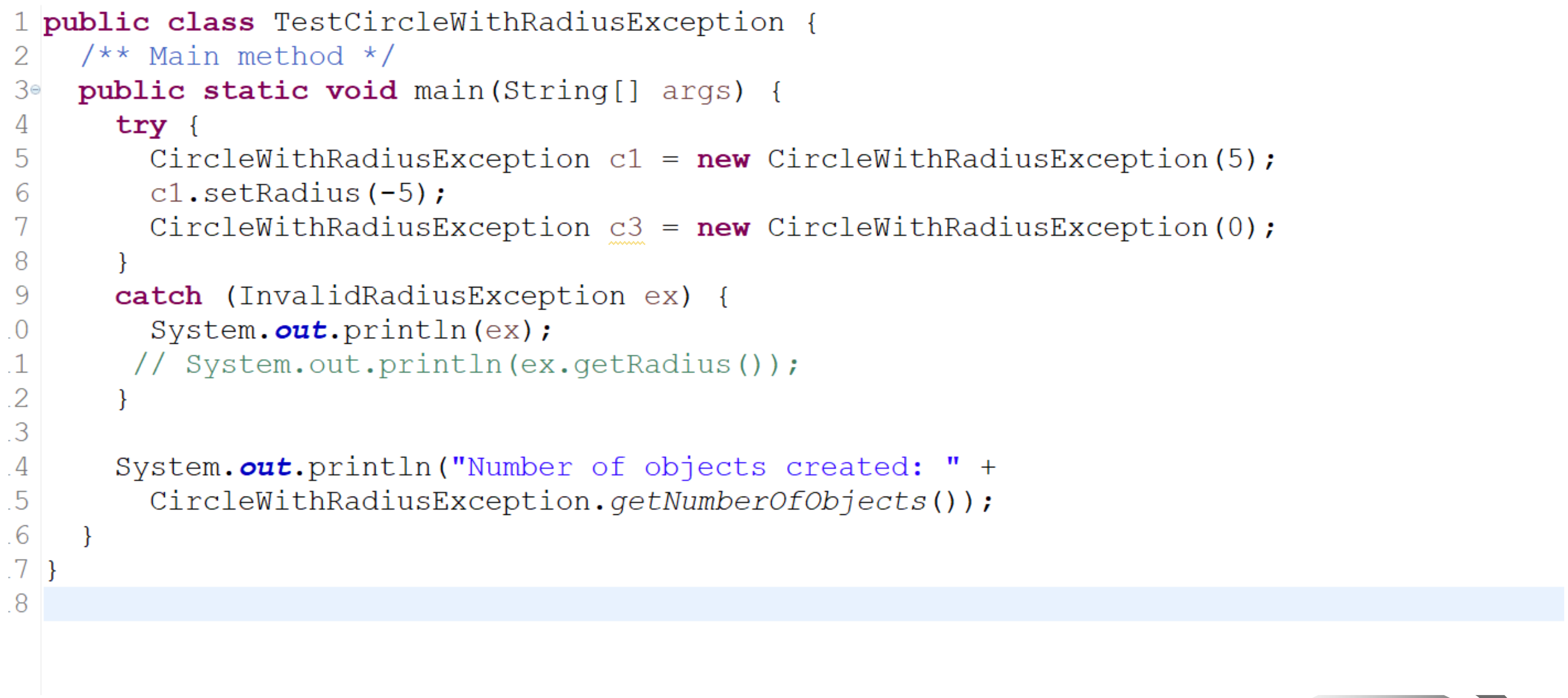
```java
public class InvalidRadiusException extends Exception {
  private double radius;

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

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



```java
public class CircleWithRadiusException {
  /** The radius of the circle */
```



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



```java
public class TestCircleWithRadiusException {
  /** Main method */
  public static void main(String[] args) {
    try {
      CircleWithRadiusException c1 = new CircleWithRadiusException(5);
      c1.setRadius(-5);
      CircleWithRadiusException c3 = new CircleWithRadiusException(0);
    }
    catch (InvalidRadiusException ex) {
      System.out.println(ex);
     // System.out.println(ex.getRadius());
    }

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

Assertions

An assertion is a Java statement that enables you to assert an assumption about your program.

An assertion contains a Boolean expression that should be true during program execution.

Assertions can be used to **assure** program correctness and avoid logic errors.

Companion Website

Declaring Assertions

An *assertion* is declared using the new Java keyword assert in JDK 1.4 as follows:

assert *assertion;* or
assert *assertion : detailMessage;*

where **assertion** is a Boolean expression and *detailMessage* is a primitive-type or an Object value.

Companion Website

Executing Assertions

When an assertion statement is executed, Java evaluates the assertion. If it is false, an AssertionError will be thrown. The **AssertionError** class has a no-arg **constructor** and seven overloaded single-argument **constructors** of type int, long, float, double, boolean, char, and Object.

For the first assert statement with no detail message, the no-arg constructor of AssertionError is used. For the second assert statement with a detail message, an appropriate AssertionError constructor is used to match the data type of the message. Since **AssertionError is a subclass of Error**, when an assertion becomes false, the program displays a message on the console and exits.

Companion Website

Executing Assertions Example

```
public class AssertionDemo {
  public static void main(String[] args) {
    int i; int sum = 0;
    for (i = 0; i < 10; i++) {
      sum += i;
    }
    assert i == 10;
    assert sum > 10 && sum < 5 * 10 : "sum is " + sum;
  }
}
```

Companion Website

Compiling Programs with Assertions

Since assert is a new Java keyword introduced in JDK 1.4, you have to compile the program using a JDK 1.4 compiler. Furthermore, you need to include the switch –source 1.4 in the compiler command as follows:

```
javac –source 1.4 AssertionDemo.java
```

NOTE: If you use JDK 1.5, there is **no need** to use the –source 1.4 option in the command.

Companion Website

Running Programs with Assertions

By default, the assertions are disabled at runtime. To enable it, use the switch –enableassertions, or –ea for short, as follows:

```
java –ea AssertionDemo
```

Assertions can be selectively enabled or disabled at class level or package level. The disable switch is –disableassertions or –da for short. For example, the following command enables assertions in package package1 and disables assertions in class Class1.

```
java –ea:package1 –da:Class1 AssertionDemo
```

Companion Website

Using Exception Handling or Assertions

- Assertion should not be used to replace exception handling.
- Exception handling deals with unusual circumstances during program execution.
- Assertions are to assure the correctness of the program.
- Exception handling addresses robustness and assertion addresses correctness.
- Like exception handling, assertions are not used for normal tests, but for internal consistency and validity checks. Assertions are checked at runtime and can be turned on or off at startup time.

Companion Website

Using Exception Handling or Assertions, cont.

Do not use assertions for argument checking in public methods.

Valid arguments that may be passed to a public method are considered to be part of the method's contract. **The contract must always be obeyed whether assertions are enabled or disabled.**

For example, the following code in the Circle class should be rewritten using exception handling.

```
public void setRadius(double newRadius) {
  assert newRadius >= 0;
  radius =  newRadius;
}
```

Assertion有可能disabled

Companion Website

Using Exception Handling or Assertions, cont.

Use assertions to reaffirm assumptions.

This gives you more confidence to assure correctness of the program.
A common use of assertions is to replace assumptions with assertions in the code.

Using Exception Handling or Assertions, cont.

Another good use of assertions is place assertions in a switch statement without a default case.

For example,

```
switch (month) {
  case 1: ... ; break;
  case 2: ... ; break;
  ...
  case 12: ... ; break;
  default: assert false : "Invalid month: " + month
}
```

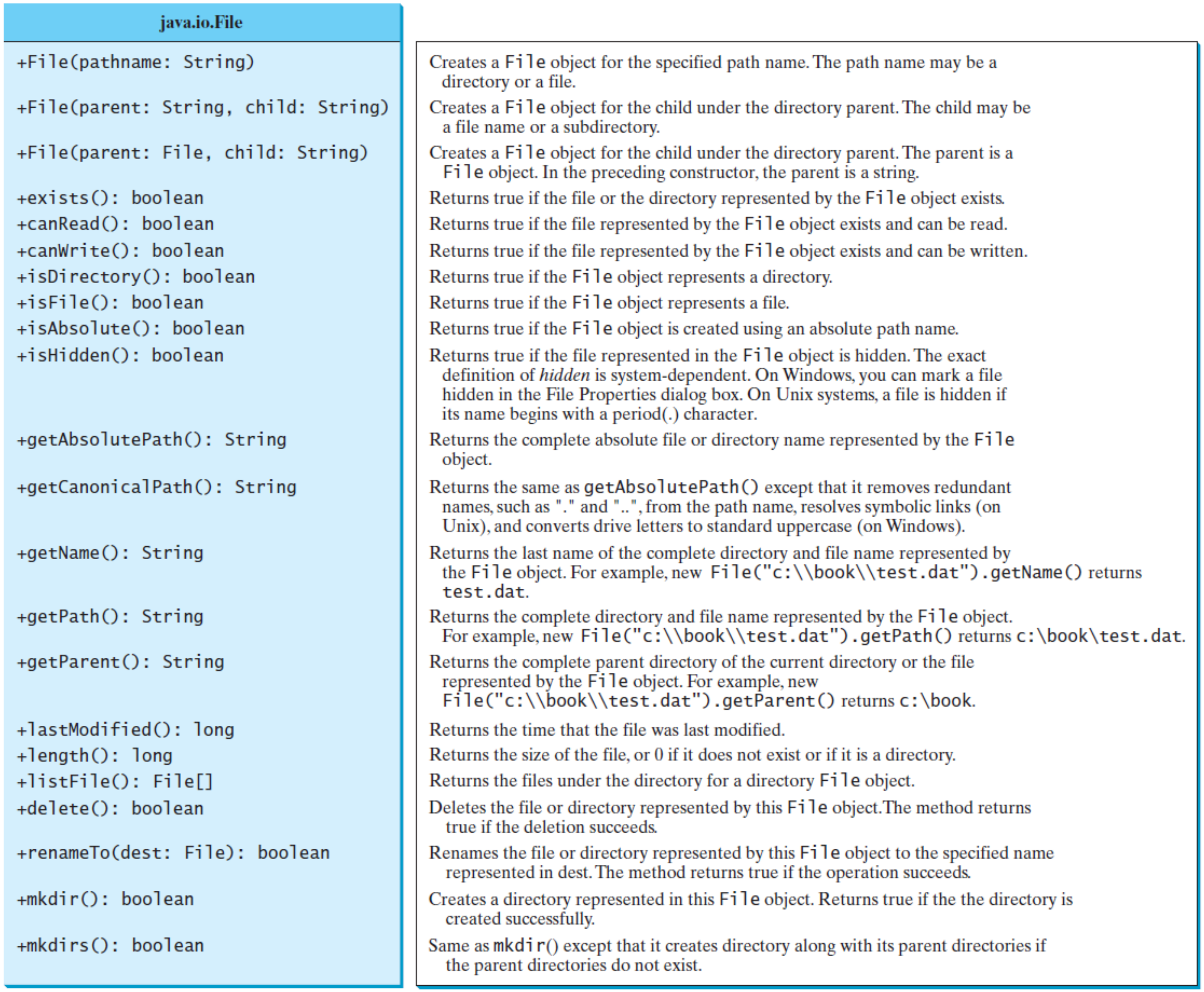
The File Class

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.

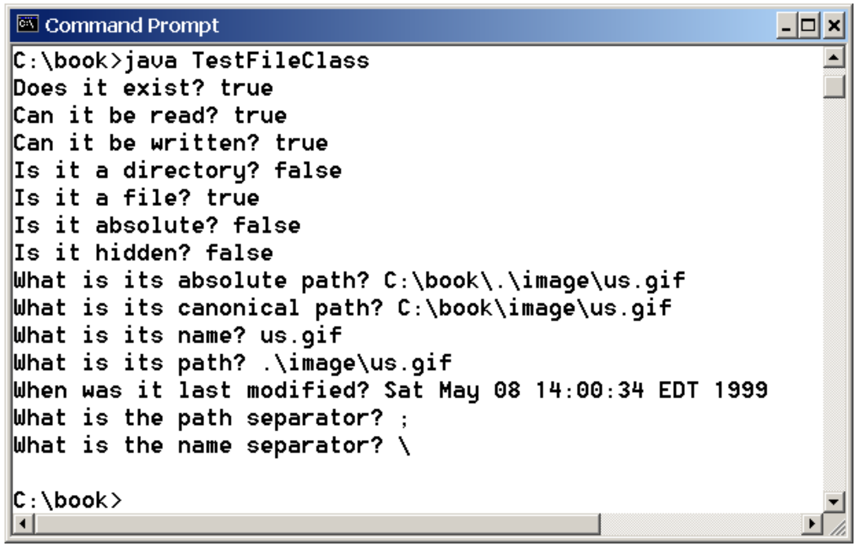
Obtaining file properties and manipulating file



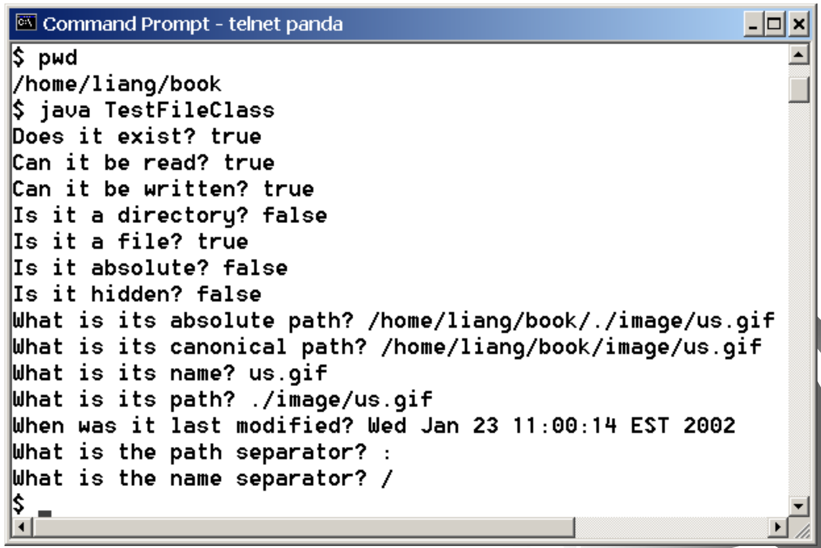
java.io.File	
+File(pathname: String)	Creates a File object for the specified path name. The path name may be a directory or a file.
+File(parent: String, child: String)	Creates a File object for the child under the directory parent. The child may be a file name or a subdirectory.
+File(parent: File, 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.



```
C:\book>java TestFileClass
Does it exist? true
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
What is its absolute path? C:\book\.\image\us.gif
What is its canonical path? C:\book\image\us.gif
What is its name? us.gif
What is its path? .\image\us.gif
When was it last modified? Sat May 08 14:00:34 EDT 1999
What is the path separator? ;
What is the name separator? \

C:\book>
```



```
$ pwd
/home/liang/book
$ java TestFileClass
Does it exist? true
Can it be read? true
Can it be written? true
Is it a directory? false
Is it a file? true
Is it absolute? false
Is it hidden? false
What is its absolute path? /home/liang/book/./image/us.gif
What is its canonical path? /home/liang/book/image/us.gif
What is its name? us.gif
What is its path? ./image/us.gif
When was it last modified? Wed Jan 23 11:00:14 EST 2002
What is the path separator? :
What is the name separator? /
$ _
```

TestFileClass

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.

Readers, Writers, and Streams

Two ways to store data: text and binary format.

Text format: human-readable form, as a sequence of characters.

E.g. Integer 12,345 stored as characters '1' '2' '3' '4' '5'.

More convenient for humans: easier to produce input and to check output.

Readers and **writers** handle data in text form.

Binary format: data items are represented in bytes.

E.g. Integer 12,345 stored as sequence of four bytes 0 0 48 57. More compact and more efficient.

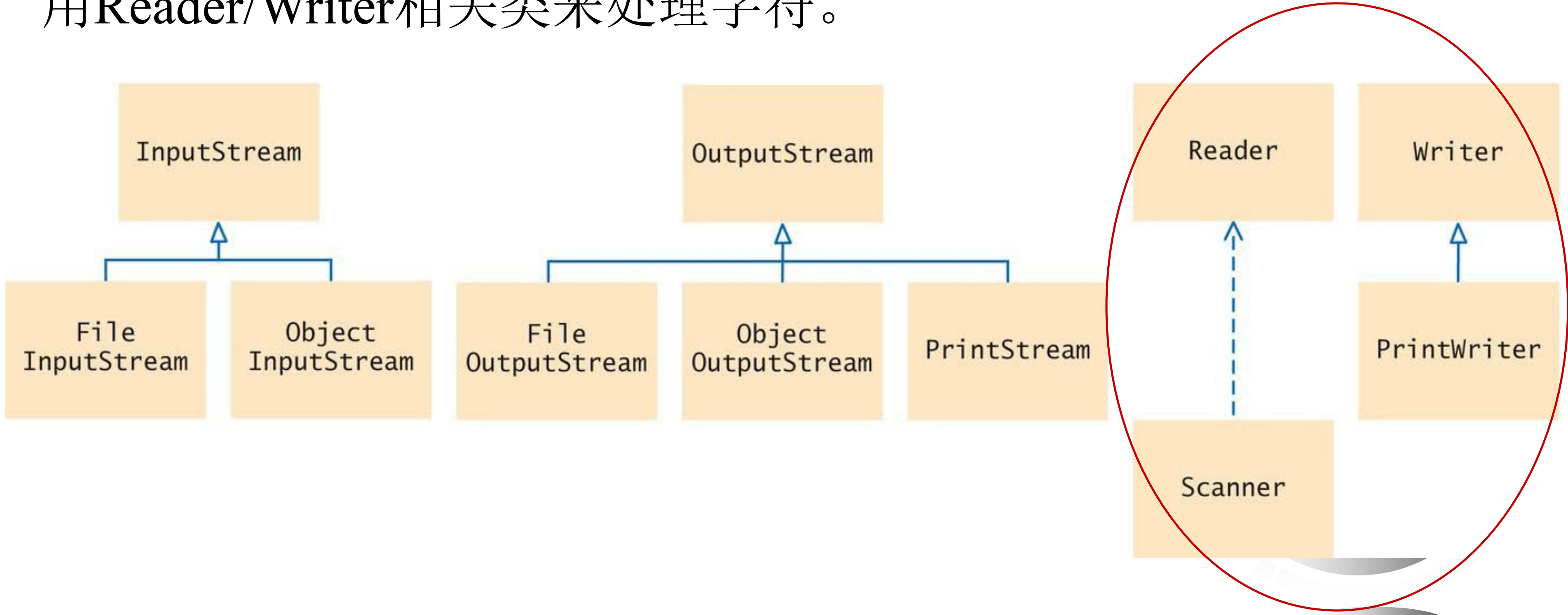
Streams handle binary data.

Java Classes for Input and Output

Java Stream相关类是用来处理字节流的,但不适合用来字符流。因为一个字节是8bit，而一个字符是16bit。

字符串是由字符组成，字符串类型天然处理的是字符而不是字节。

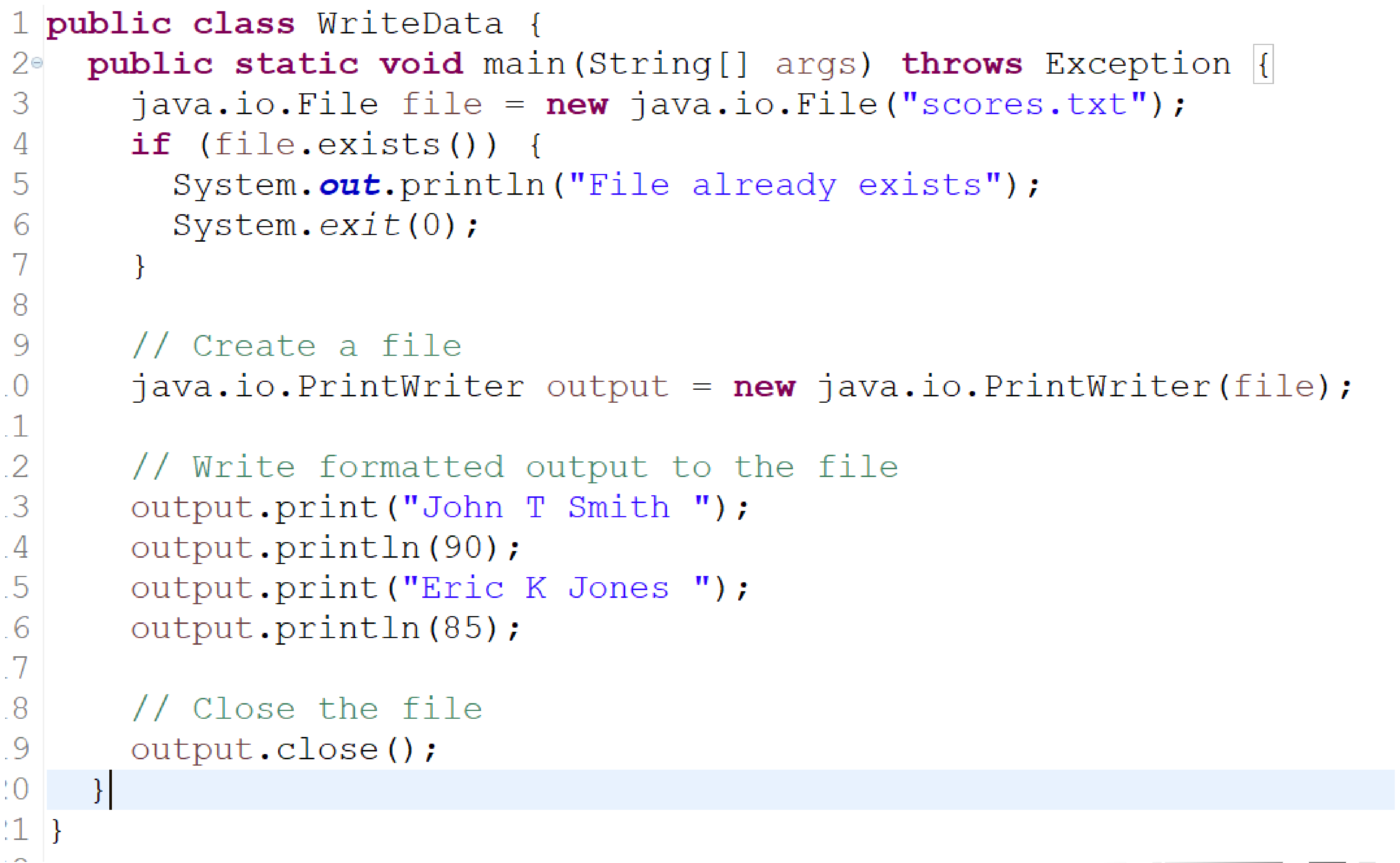
更重要的是，字节流无法知道字符集及其字符编码。Java中可以用Reader/Writer相关类来处理字符。



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



```java
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);

    // 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();
  }
}
```

Try-with-resources

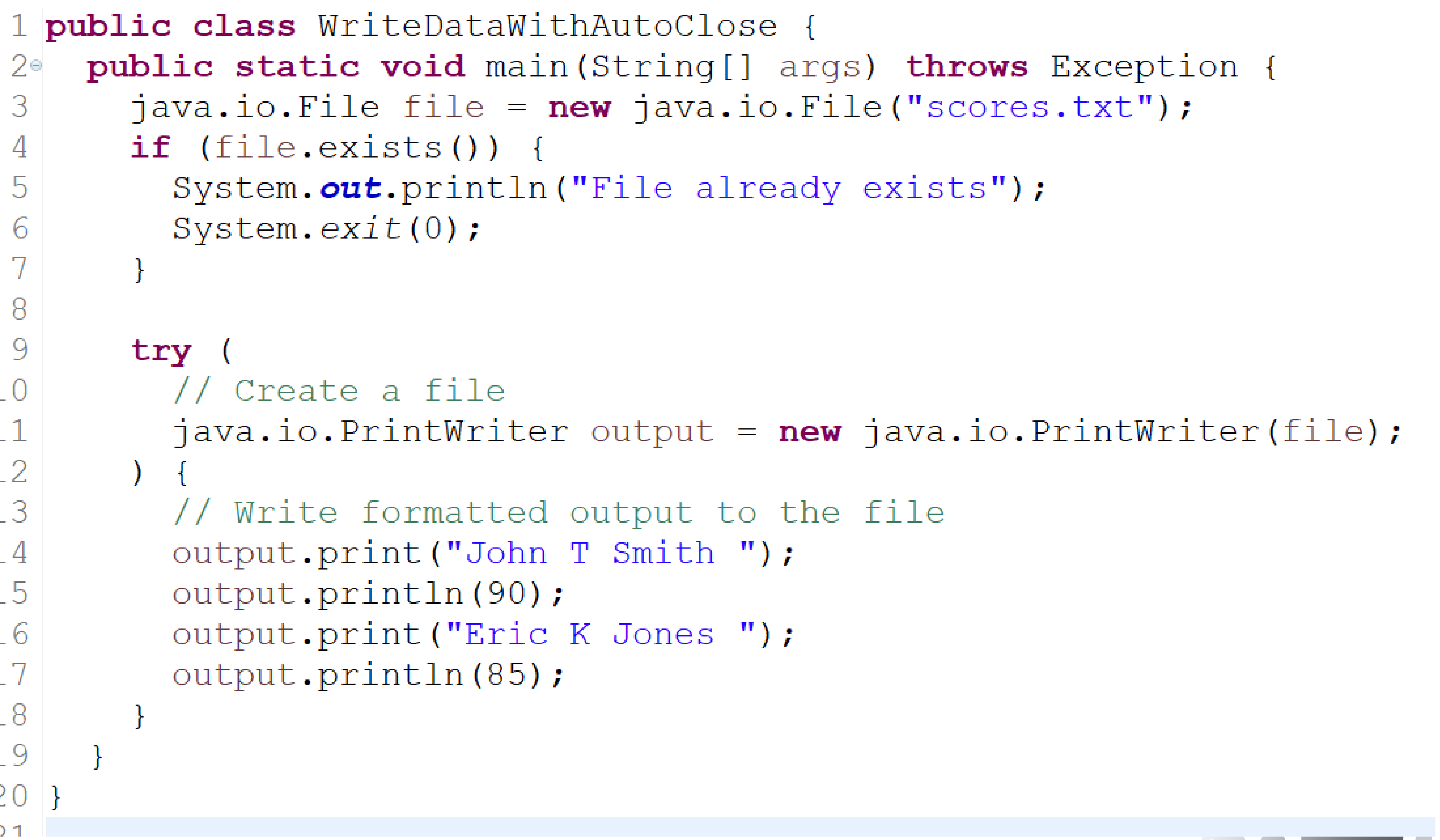
Programmers often forget to close the file. JDK 7 provides the followings new try-with-resources syntax that automatically closes the files.

try (declare and create resources) {

  Use the resource to process the file;

}

WriteDataWithAutoClose Run



```java
public class WriteDataWithAutoClose {
  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);
    }

    try (
      // 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);
    }
  }
}
```

Reading Data Using <u>Scanner</u>

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

CAUTION

```
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
PrintWriter(Writer wr)
PrintWriter(Writer wr, boolean autoFlush)
PrintWriter(File file)
PrintWriter(File file, String csn)
PrintWriter(String fileName)
PrintWriter(String fileName, String csn)
```

- 可以指定字符集，否则会使用运行这个程序的机器的默认编码，可能在不同的机器上有不同的表现。如
- PrintWriter out = new PrintWriter(“myfile.txt”, “UTF-8”);

CAUTION

- Scanner也是一样，可以设定字符编码
- Scanner in = new Scanner(Paths.get(“myfile.txt”),”UTF-8”);
- Scanner in = new Scanner(“myfile.txt”); //ERROR?
- 构造了一个带字符串参数的Scanner，将字符串解释为数据，而不是文件名。
- Scanner in = new Scanner(new File(“myfile.txt”),”UTF-8”);

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
```

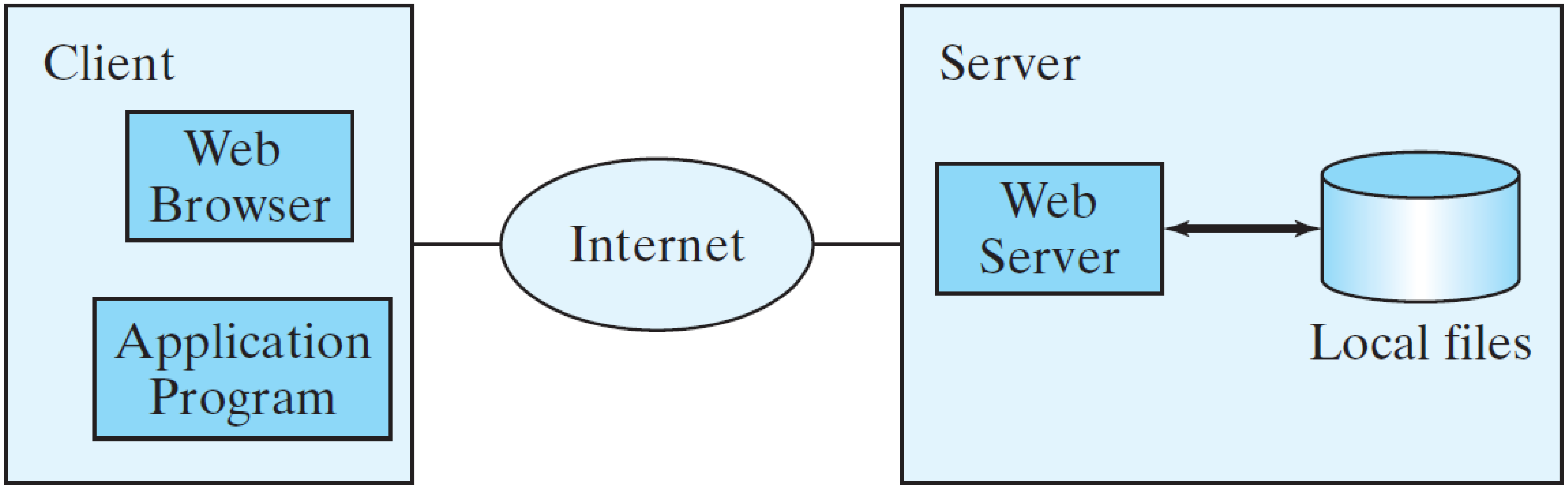
replaces all the occurrences of StringBuilder by StringBuffer in FormatString.java and saves the new file in t.txt.

ReplaceText   Run

```
try (
  // Create input and output files
  Scanner input = new Scanner(sourceFile);
  PrintWriter output = new PrintWriter(targetFile);
) {
  while (input.hasNext()) {
    String s1 = input.nextLine();
    String s2 = s1.replaceAll(args[2], args[3]);
    output.println(s2);
  }
}
```

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:

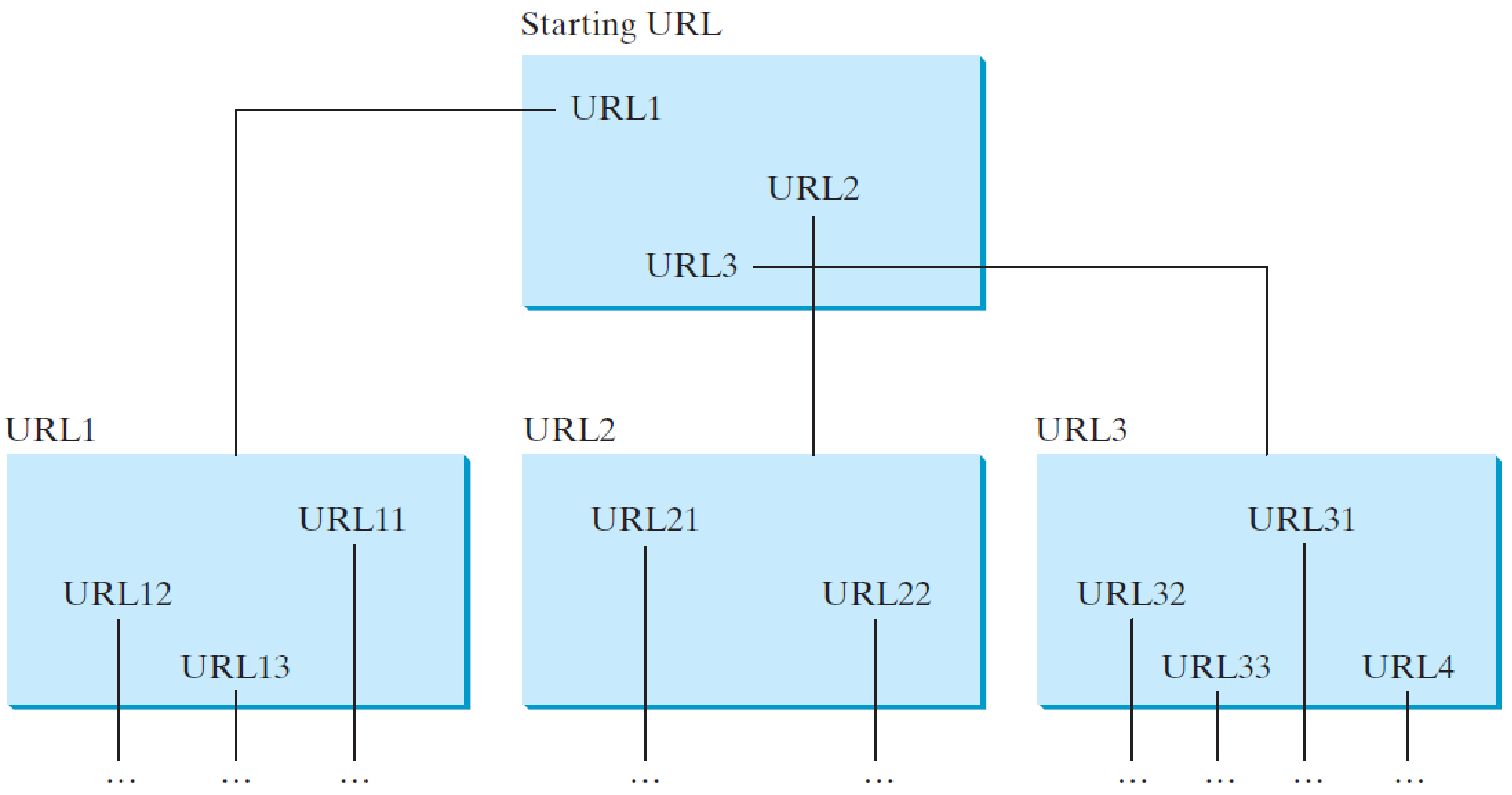
Scanner input = **new** Scanner(url.openStream());

ReadFileFromURL

Run

Case Study: Web Crawler

This case study develops a program that travels the Web by following hyperlinks.



Case Study: Web Crawler

The program follows the URLs to traverse the Web. To avoid that each URL is traversed only once, the program maintains two lists of URLs. One list stores the URLs pending for traversing and the other stores the URLs that have already been traversed. The algorithm for this program can be described as follows:

Case Study: Web Crawler

```
Add the starting URL to a list named listOfPendingURLs;
while listOfPendingURLs is not empty {
    Remove a URL from listOfPendingURLs;
    if this URL is not in listOfTraversedURLs {
      Add it to listOfTraversedURLs;
      Display this URL;
      Exit the while loop when the size of S is equal to 100.
      Read the page from this URL and for each URL contained in the page {
        Add it to listOfPendingURLs if it is not is listOfTraversedURLs;
      }
  }
}
```

WebCrawler

Run