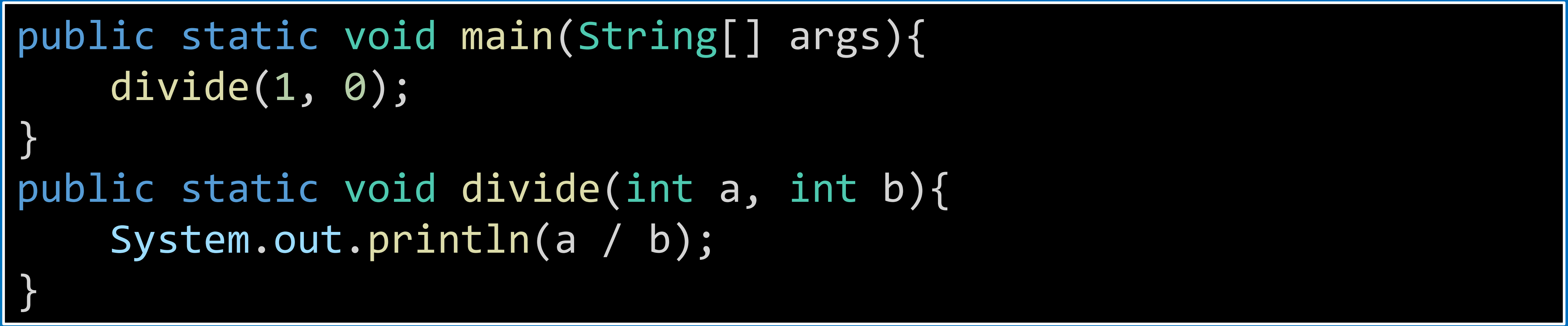
Chapter 09 Exception Handling

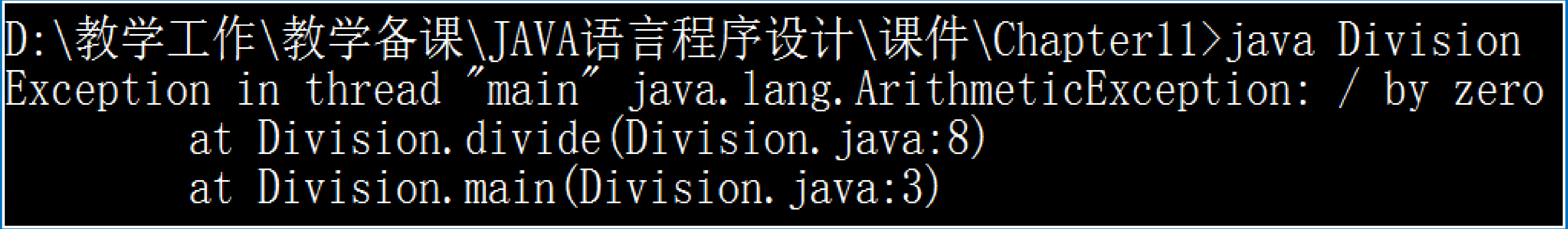
Motivations

An *exception* is an *object* that represents an *error* or *a condition that prevents execution from proceeding normally*. If the exception is not handled, the program *terminates abnormally*. How can you handle the runtime error so that the program can *continue* to run or *terminate gracefully*?

Case: Division



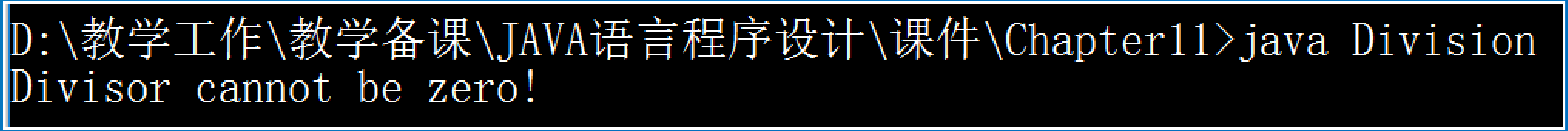
```java
public static void main(String[] args){
    divide(1, 0);
}
public static void divide(int a, int b){
    System.out.println(a / b);
}
```



```
D:\教学工作\教学备课\JAVA语言程序设计\课件\Chapter11>java Division
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Division.divide(Division.java:8)
        at Division.main(Division.java:3)
```

Case: Division with `if`

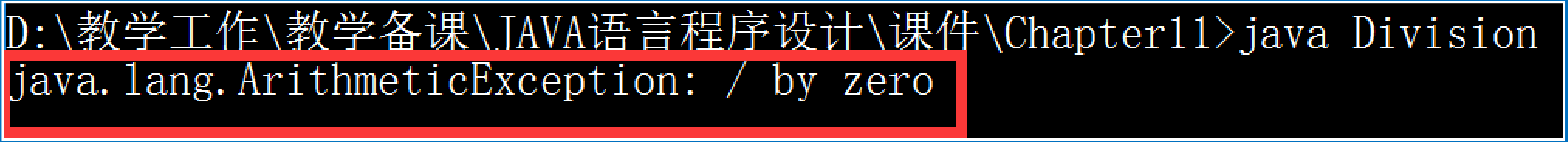
```
public static void divide(int a, int b){
    if(b != 0)
        System.out.println(a / b);
    else
        System.out.println("Divisor cannot be zero!");
}
```



```
D:\教学工作\教学备课\JAVA语言程序设计\课件\Chapter11>java Division
Divisor cannot be zero!
```

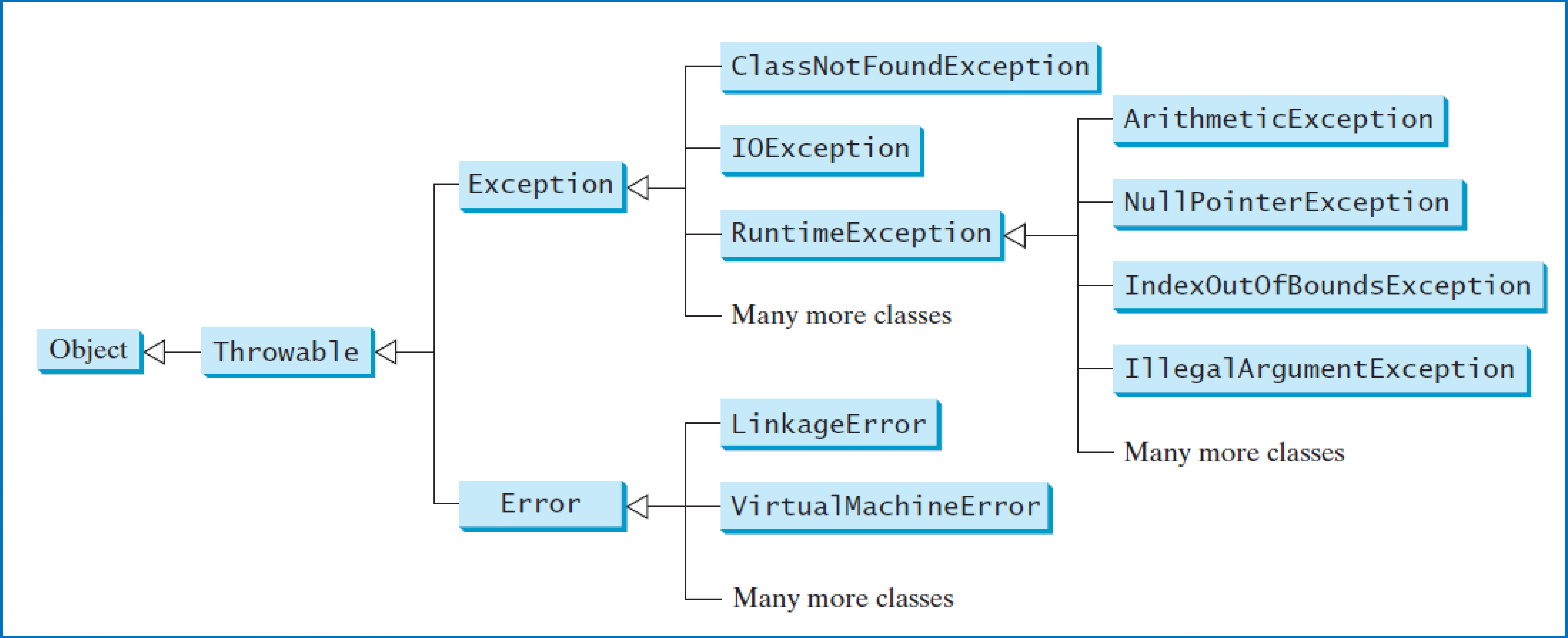
Case: Division with Exception

```java
public static void divide(int a, int b){
    try{
        System.out.println(a / b);
    }catch(ArithmeticException ae){
        System.out.println(ae.toString());
    }
}
```



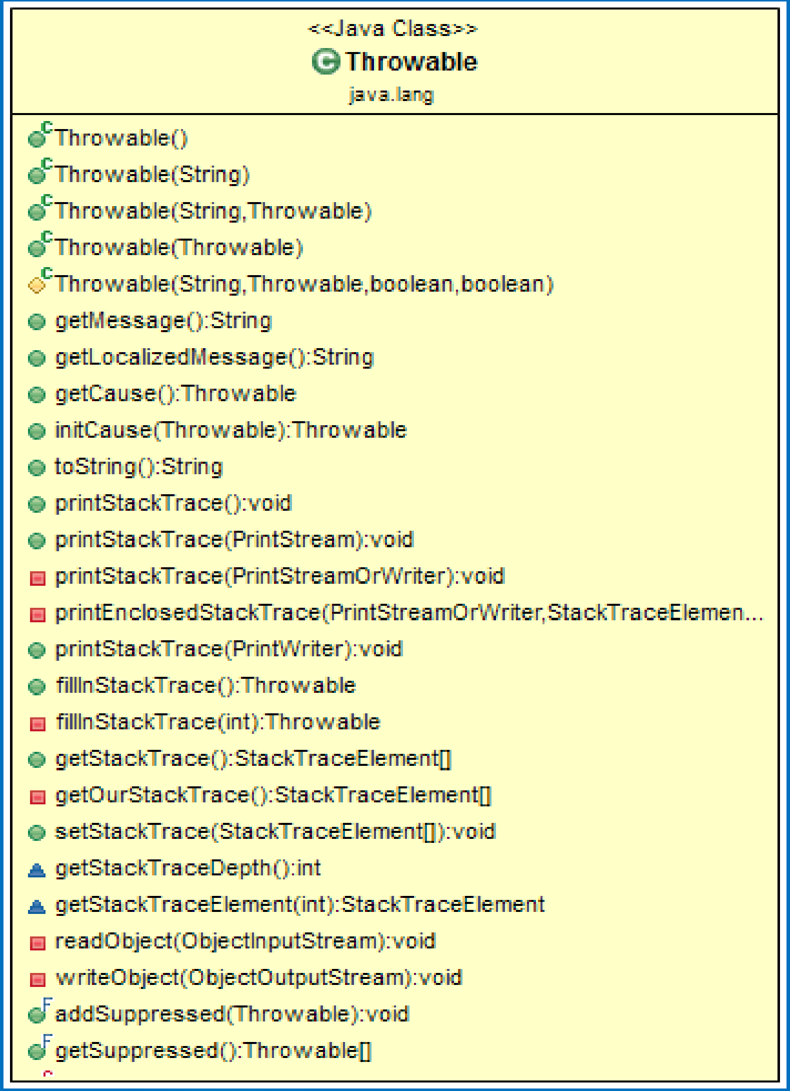
```
D:\教学工作\教学备课\JAVA语言程序设计\课件\Chapter11>java Division
java.lang.ArithmeticException: / by zero
```

Exception and Error Types



Throwable

The *Throwable* class is the superclass of all *errors* and *exceptions* in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java *throw* statement. Similarly, only this class or one of its subclasses can be the argument type in a *catch* clause.



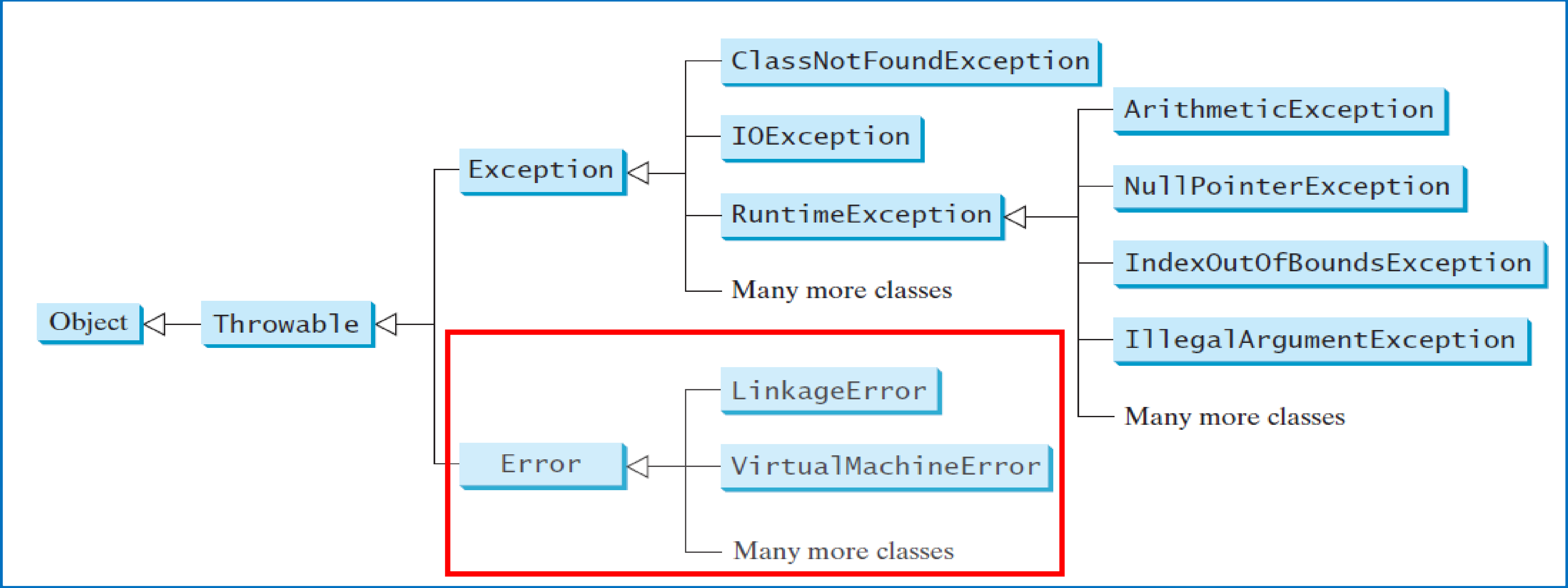
```
<<Java Class>>
Throwable
java.lang

Throwable()
Throwable(String)
Throwable(String,Throwable)
Throwable(Throwable)
Throwable(String,Throwable,boolean,boolean)
getMessage():String
getLocalizedMessage():String
getCause():Throwable
initCause(Throwable):Throwable
toString():String
printStackTrace():void
printStackTrace(PrintStream):void
printStackTrace(PrintStreamOrWriter):void
printEnclosedStackTrace(PrintStreamOrWriter,StackTraceElemen...
printStackTrace(PrintWriter):void
fillInStackTrace():Throwable
fillInStackTrace(int):Throwable
getStackTrace():StackTraceElement[]
getOurStackTrace():StackTraceElement[]
setStackTrace(StackTraceElement[]):void
getStackTraceDepth():int
getStackTraceElement(int):StackTraceElement
readObject(ObjectInputStream):void
writeObject(ObjectOutputStream):void
addSuppressed(Throwable):void
getSuppressed():Throwable[]
```

Throwable Example

```
public class Test {
    public static void main(String[] args) {
        try {
            throwException(null);
        } catch (Throwable e) {
            System.out.println(e.getMessage());
            System.out.println(e.getLocalizedMessage());
            System.out.println(e.getClass());
        }
    }
    private static void throwException(String str) throws Throwable {
        if (str == null) {
            throw new Throwable("String is null.");
        }
    }
}
```

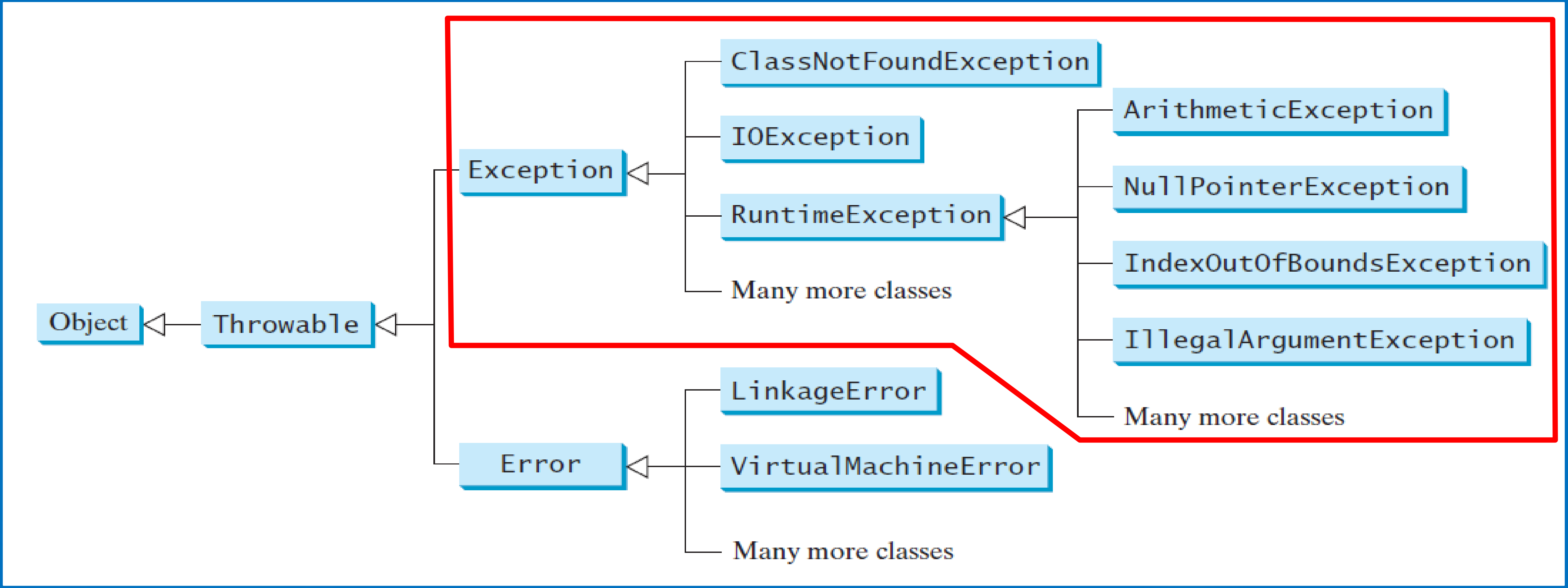
System Errors

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.



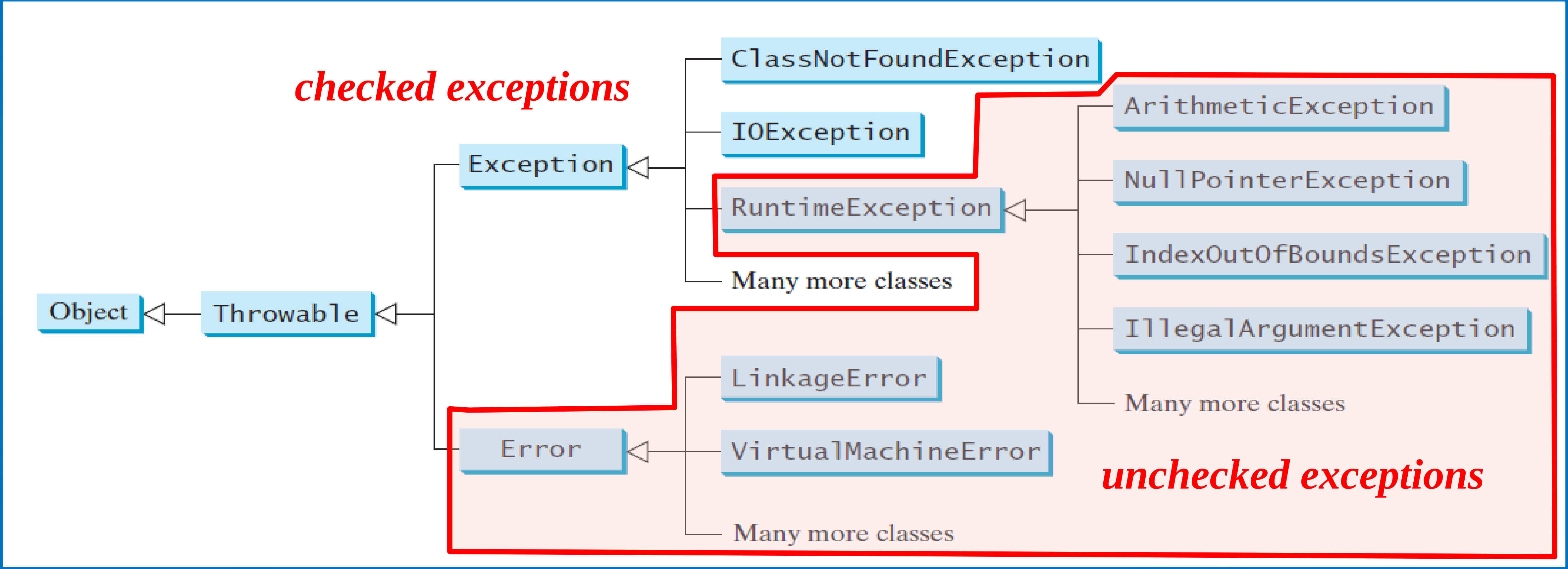
Exceptions

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



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.

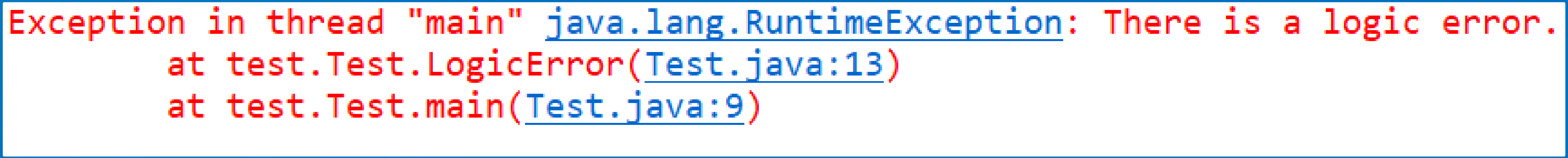


Unchecked Exceptions

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.

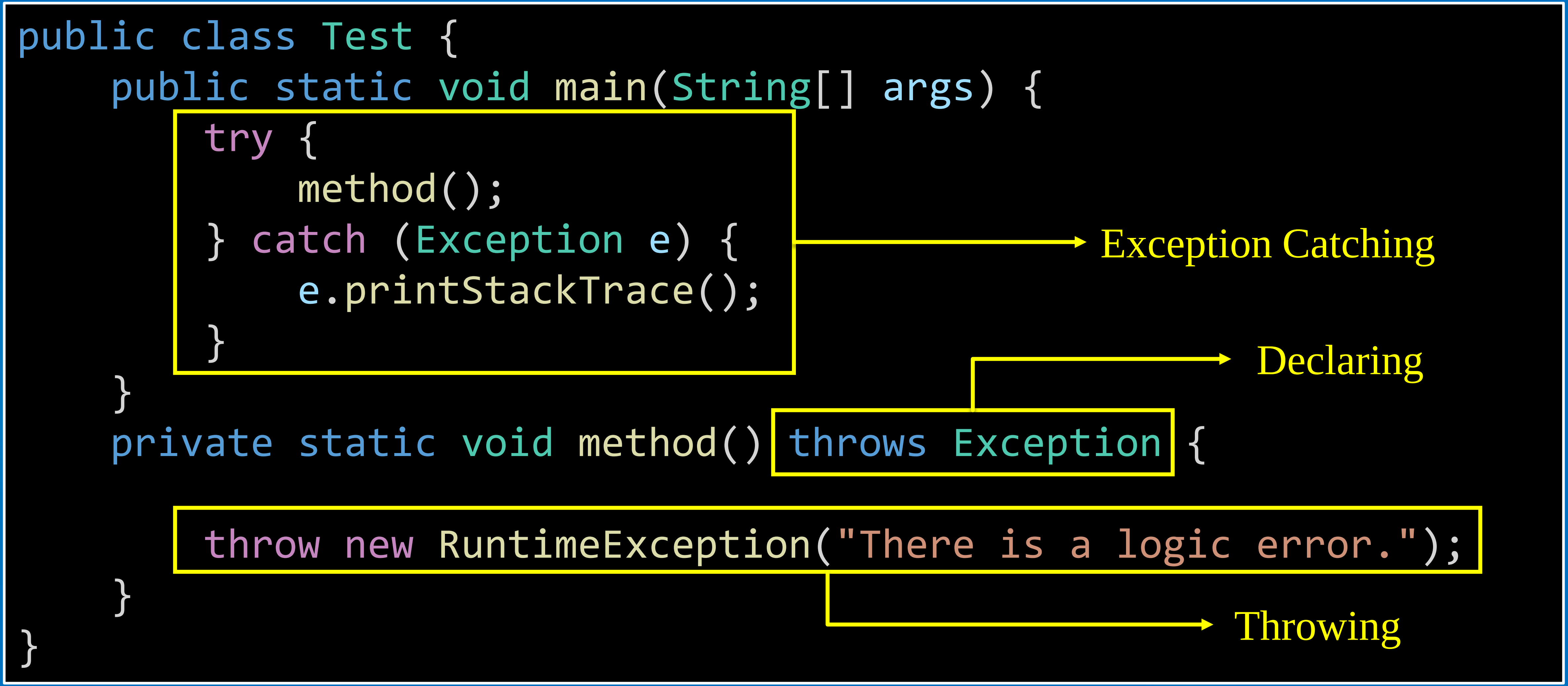
Unchecked Exception Example

```java
public class Test {
    public static void main(String[] args) {
        LogicError();
    }

    private static void LogicError() throws RuntimeException {
        throw new RuntimeException("There is a logic error.");
    }
}
```



```
Exception in thread "main" java.lang.RuntimeException: There is a logic error.
        at test.Test.LogicError(Test.java:13)
        at test.Test.main(Test.java:9)
```

Declaring, Throwing, and Catching Exceptions



```
public class Test {
    public static void main(String[] args) {
        try {
            method();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    private static void method() throws Exception {

        throw new RuntimeException("There is a logic error.");
    }
}
```

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 Exception();

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

Throwing Exceptions

```
public static void main(String[] args){
    try{
        throw new Exception();
        System.out.println("Do something after throwing!");
    }catch(Exception e){

    }
    System.out.println("Do something after catching!");
}
```

Can run

Cannot run

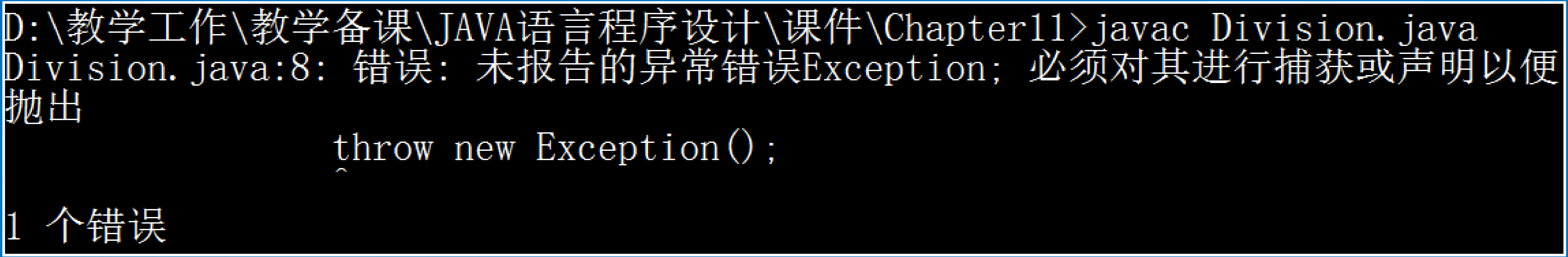
Declaring Exceptions

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
abstract class C {
    abstract void method() throws Exception;
}
interface A{
    void method() throws Exception;
}
```

Checked Exception Without Declaring

```
public static void method(){
    throw new Exception();
}
```



```
D:\教学工作\教学备课\JAVA语言程序设计\课件\Chapter11>javac Division.java
Division.java:8: 错误: 未报告的异常错误Exception; 必须对其进行捕获或声明以便
抛出
                throw new Exception();
                ^
1 个错误
```

```
// Correct the program:
public static void method() throws Exception{
    throw new Exception();
}
```

Catching and Handling Exceptions

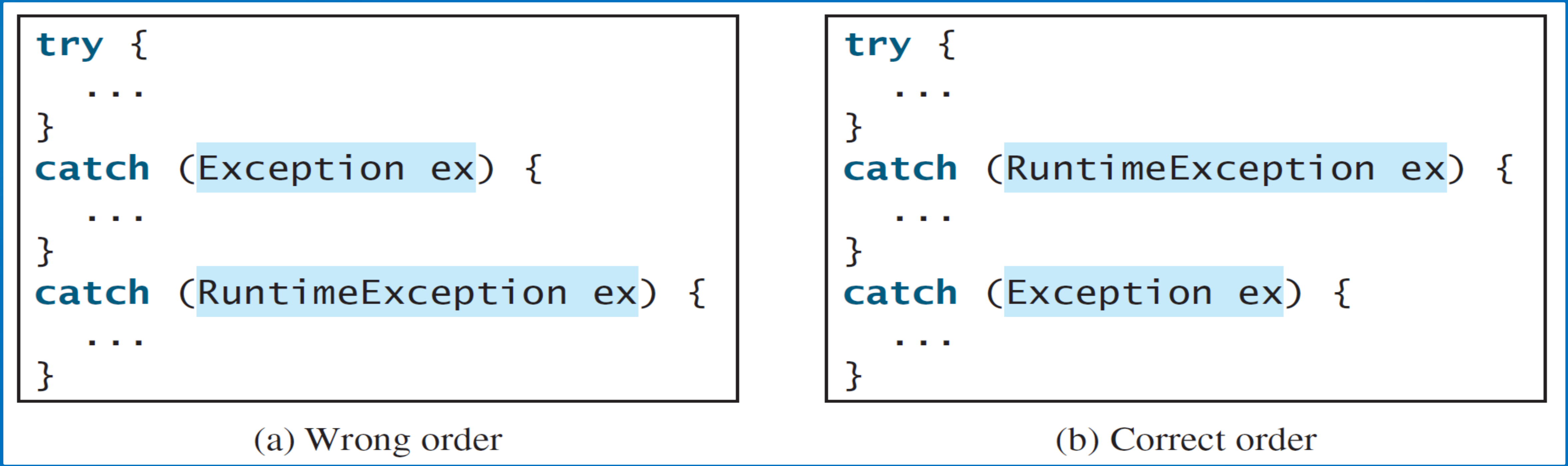
```
try {
    statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
    handler for exception1;
}
catch (Exception2 exVar2) {
    handler for exception2;
}
...
catch (ExceptionN exVar3) {
    handler for exceptionN;
}
```

Multiple Exceptions in the Same Catch Clause

```
public class Test {
    public static void main(String[] args) {
        try {
            method(1);
        }catch(ExceptionA | ExceptionB e){
            e.printStackTrace();
        }
    }
    public static void method(int i) throws ExceptionA, ExceptionB {
        if(i == 1) throw new ExceptionA();
        else throw new ExceptionB();
    }
}
class ExceptionA extends Exception{}
class ExceptionB extends Exception{}
```

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).



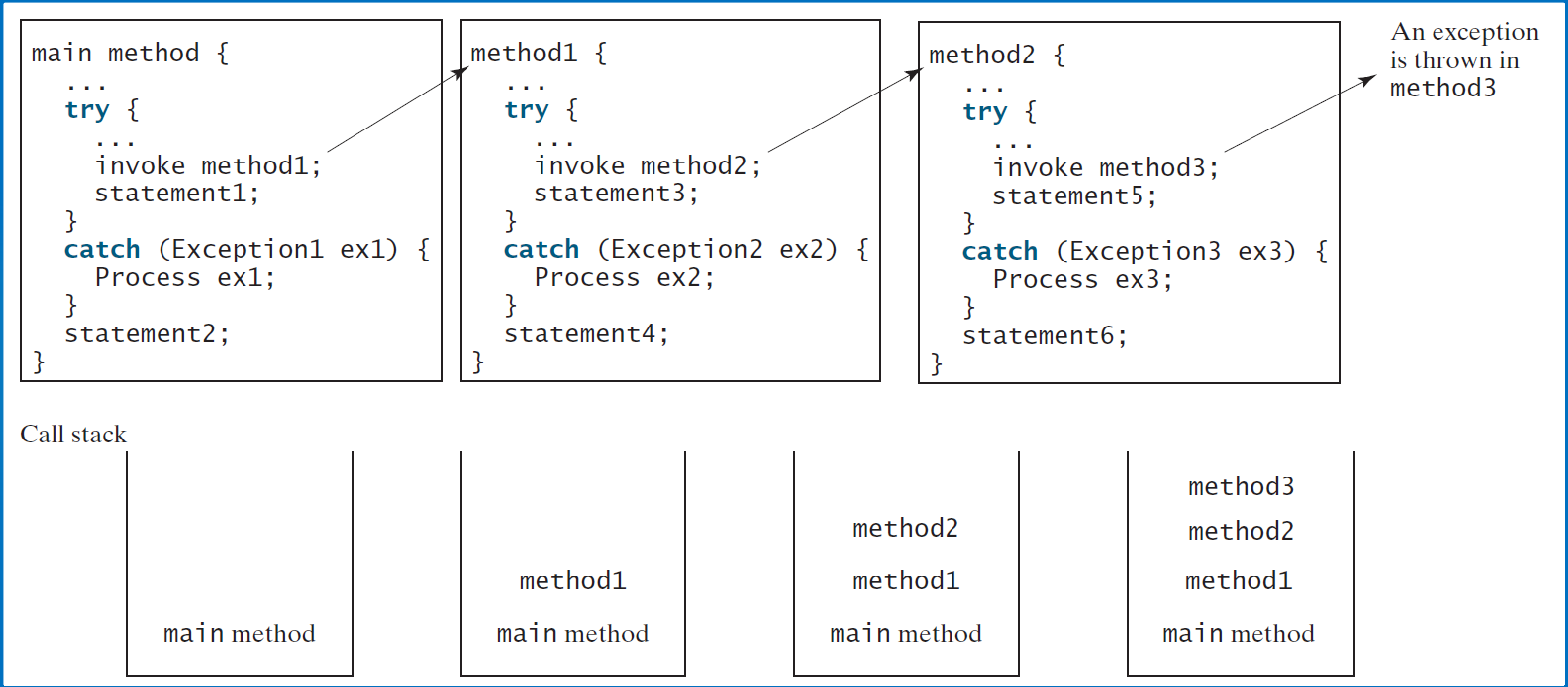
```
try {
   ...
}
catch (Exception ex) {
   ...
}
catch (RuntimeException ex) {
   ...
}
```

(a) Wrong order

```
try {
   ...
}
catch (RuntimeException ex) {
   ...
}
catch (Exception ex) {
   ...
}
```

(b) Correct order

Catching Exceptions and Stack Trace



```
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;
}
```

Super-Exception and Sub-Exception Catching

```
public class Test {
    public static void main(String[] args) {
        try {
            throw new SubException();
        }catch(SubException e){
            e.printStackTrace();
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
class SubException extends Exception{}
```

Sub-Exception cannot be after the Super-Exception

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.

CircleWithException

TestCircleWithException

Rethrowing Exceptions

```
public class Test {
    public static void main(String[] args) throws Exception {
        try {
            method(1);
        }catch(Exception e){
            if(e instanceof SubException) {
                e.printStackTrace();
            }else {
                throw e; // Rethrowing
            }
        }
    }
    public static void method(int i) throws Exception {
        if(i > 0) throw new SubException();
        else throw new Exception();
    }
}
class SubException extends Exception{}
```

The finally Clause

```
//  Suppose no exceptions in the statements
try {
    // statements; 1
}
catch(TheException ex) {
    // handling ex;
}
finally {
    // finalStatements; 2
}
Next statement; 3
```

The finally Clause

```
//  Suppose an exception of type Exception1 is thrown in statement
try {
    // statements; 1
}
catch(TheException ex) {
    // handling ex; 2
}
finally {
    // finalStatements; 3
}
Next statement; 4
```

The finally Clause

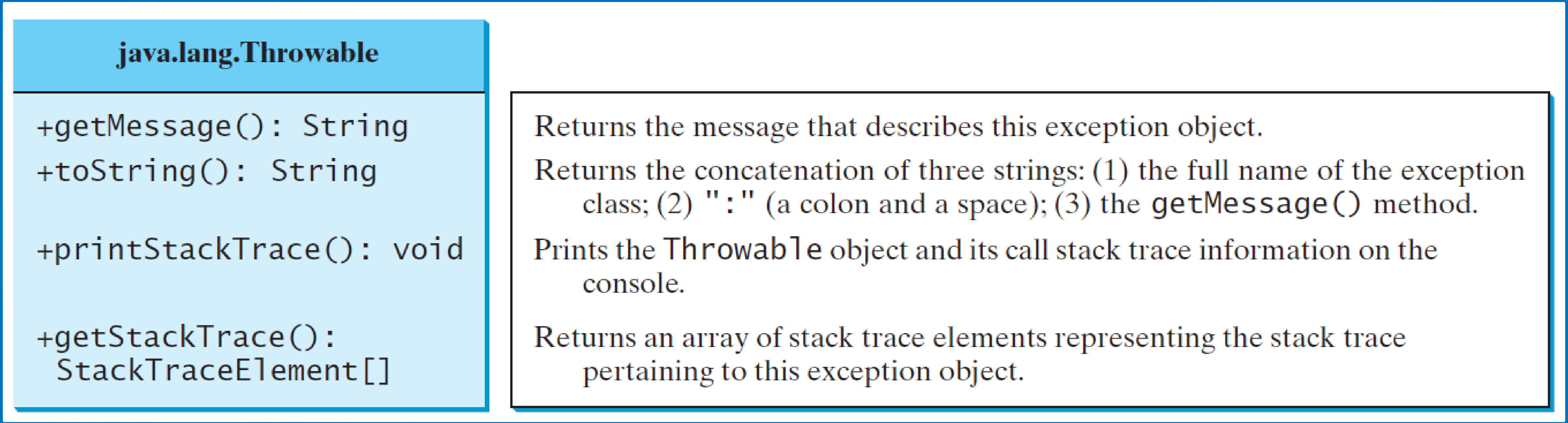
```
//  Rethrow the exception and control is transferred to the caller
try {
    // statements;  1
}
catch(TheException ex) {
    // handling ex;  2

    // rethrow ex;   4
}
finally {
    // finalStatements;  3
}
Next statement;
```

Chained Exceptions

```
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");
    }
}
```

Information of Exceptions

An exception object contains valuable information about the exception. You may use the following instance methods in the java.lang.*Throwable* class to get information regarding the exception.

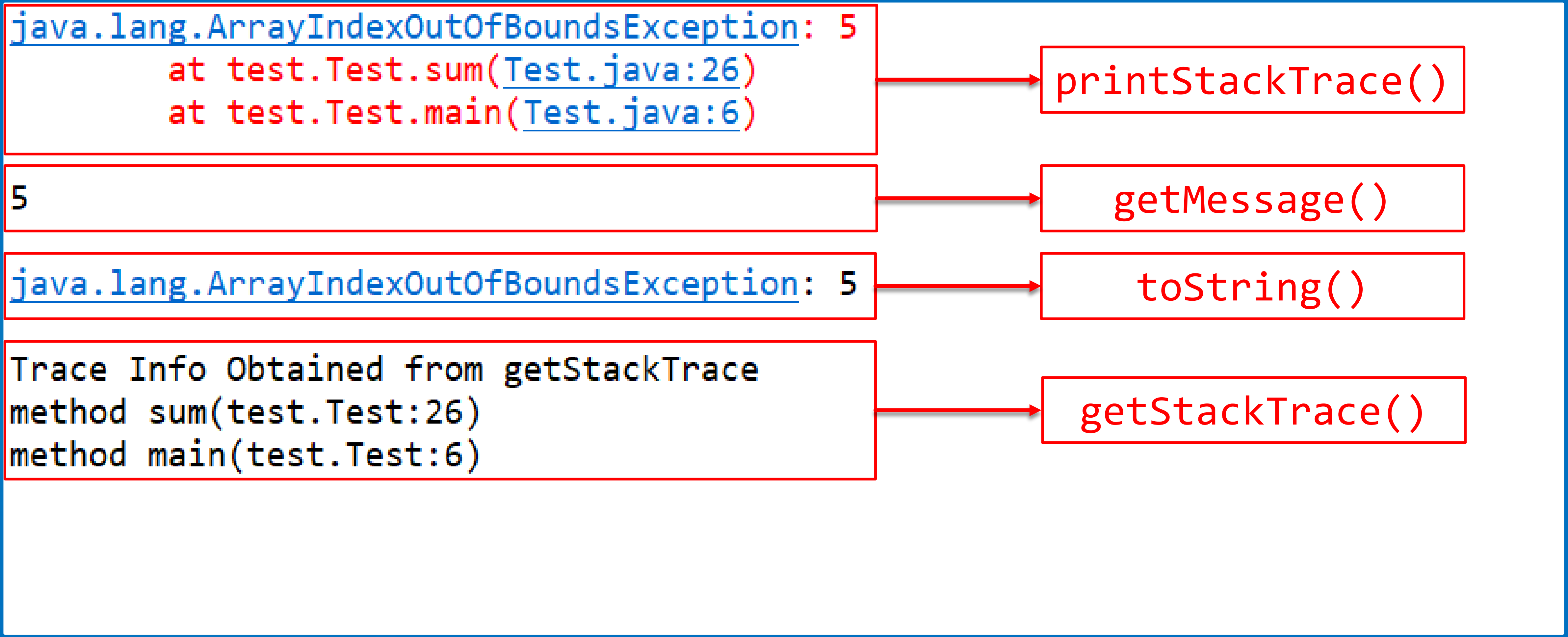


java.lang.Throwable	
+getMessage(): String	Returns the message that describes this exception object.
+toString(): String	Returns the concatenation of three strings: (1) the full name of the exception class; (2) ": " (a colon and a space); (3) the getMessage() method.
+printStackTrace(): void	Prints the 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.

Information of Exceptions

```java
public static void main(String[] args) {
    try {
        System.out.println(sum(new int[] {1, 2, 3, 4, 5}));
    }
    catch (Exception ex) {
        ex.printStackTrace();
        System.out.println("\n" + ex.getMessage());
        System.out.println("\n" + ex.toString());
        System.out.println("\nTrace Info Obtained from getStackTrace");
        StackTraceElement[] traceElements = ex.getStackTrace();
        for (int i = 0; i < traceElements.length; i++) {
            System.out.print("method " + traceElements[i].getMethodName());
            System.out.print("(" + traceElements[i].getClassName() + ":");
            System.out.println(traceElements[i].getLineNumber() + ")");
        }
    }
}
```

Information of Exceptions



```
java.lang.ArrayIndexOutOfBoundsException: 5
	at test.Test.sum(Test.java:26)
	at test.Test.main(Test.java:6)
```

```
5
```

```
java.lang.ArrayIndexOutOfBoundsException: 5
```

```
Trace Info Obtained from getStackTrace
method sum(test.Test:26)
method main(test.Test:6)
```

When to Throw 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.

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 Throw 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");
}
```

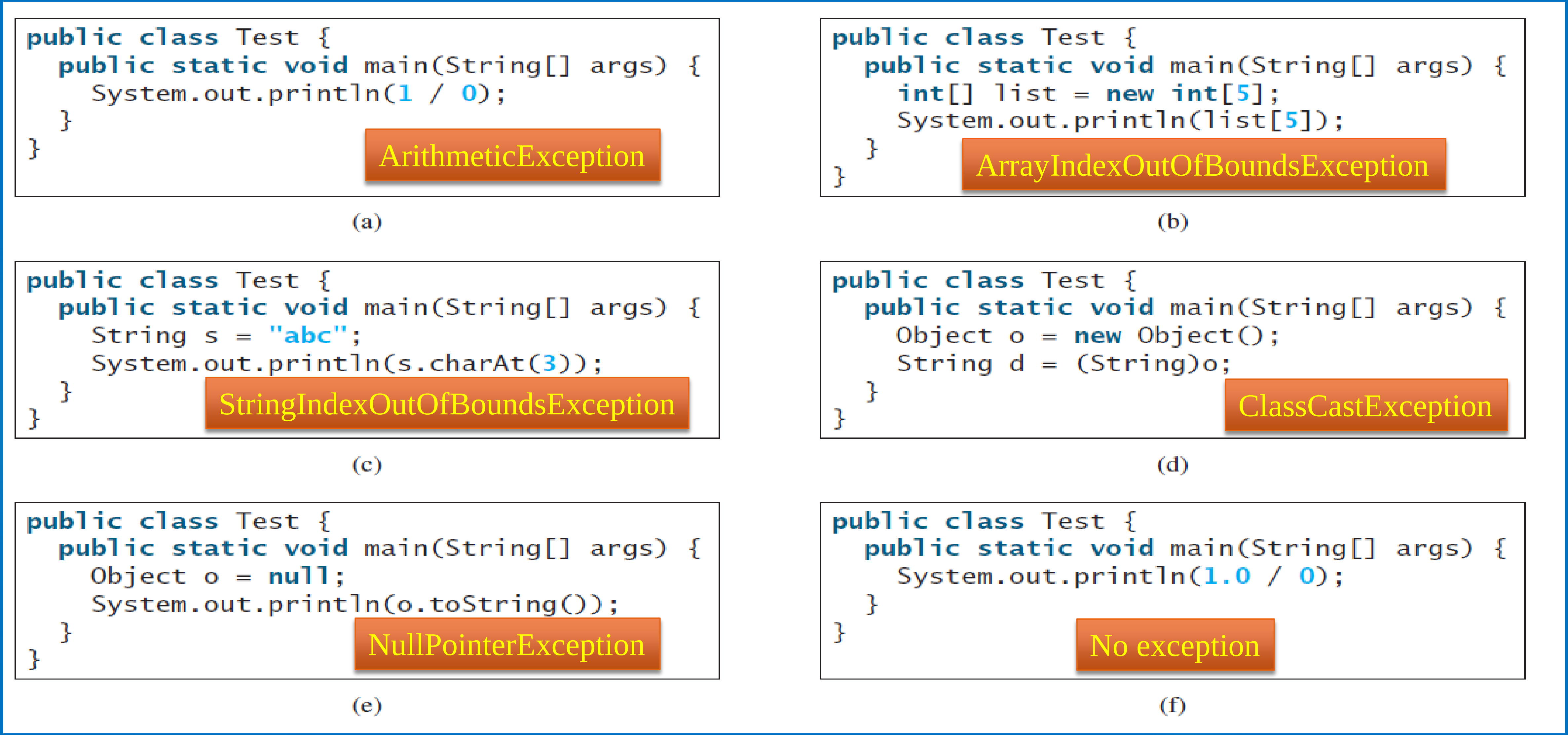
```
if (refVar != null)
  System.out.println(refVar.toString());
else
  System.out.println("refVar is null");
```

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.

InvalidRadiusException | CircleWithRadiusException | TestCircleWithRadiusException

Exercises



```
public class Test {
  public static void main(String[] args) {
    System.out.println(1 / 0);
  }
}
```

ArithmeticException

(a)

```
public class Test {
  public static void main(String[] args) {
    int[] list = new int[5];
    System.out.println(list[5]);
  }
}
```

ArrayIndexOutOfBoundsException

(b)

```
public class Test {
  public static void main(String[] args) {
    String s = "abc";
    System.out.println(s.charAt(3));
  }
}
```

StringIndexOutOfBoundsException

(c)

```
public class Test {
  public static void main(String[] args) {
    Object o = new Object();
    String d = (String)o;
  }
}
```

ClassCastException

(d)

```
public class Test {
  public static void main(String[] args) {
    Object o = null;
    System.out.println(o.toString());
  }
}
```

NullPointerException

(e)

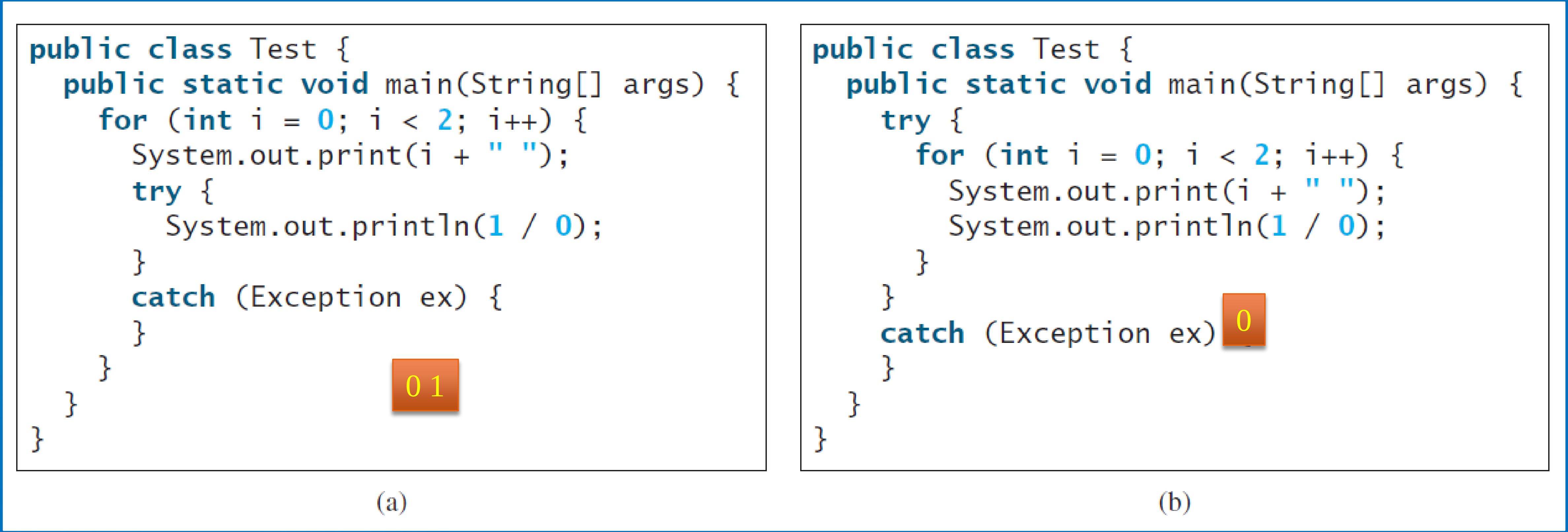
```
public class Test {
  public static void main(String[] args) {
    System.out.println(1.0 / 0);
  }
}
```

No exception

(f)

Exercises

Show the output of the following code.



```java
public class Test {
  public static void main(String[] args) {
    for (int i = 0; i < 2; i++) {
      System.out.print(i + " ");
      try {
        System.out.println(1 / 0);
      }
      catch (Exception ex) {
      }
    }
  }
}
```

0 1

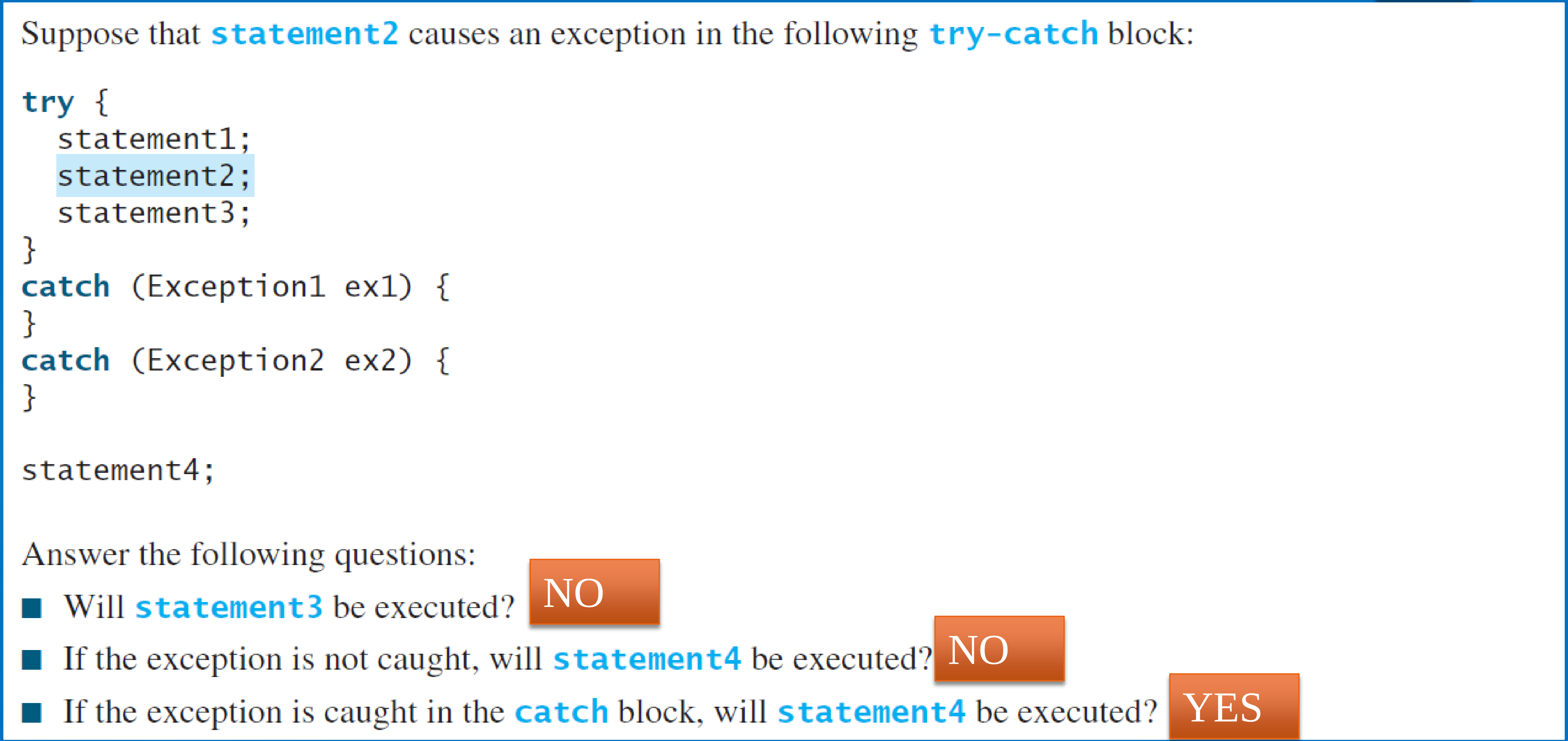
(a)

```java
public class Test {
  public static void main(String[] args) {
    try {
      for (int i = 0; i < 2; i++) {
        System.out.print(i + " ");
        System.out.println(1 / 0);
      }
    }
    catch (Exception ex)
    }
  }
}
```

0

(b)

Exercises



Suppose that `statement2` causes an exception in the following `try-catch` block:

```
try {
  statement1;
  statement2;
  statement3;
}
catch (Exception1 ex1) {
}
catch (Exception2 ex2) {
}

statement4;
```

Answer the following questions:

- Will `statement3` be executed? NO
- If the exception is not caught, will `statement4` be executed? NO
- If the exception is caught in the `catch` block, will `statement4` be executed? YES

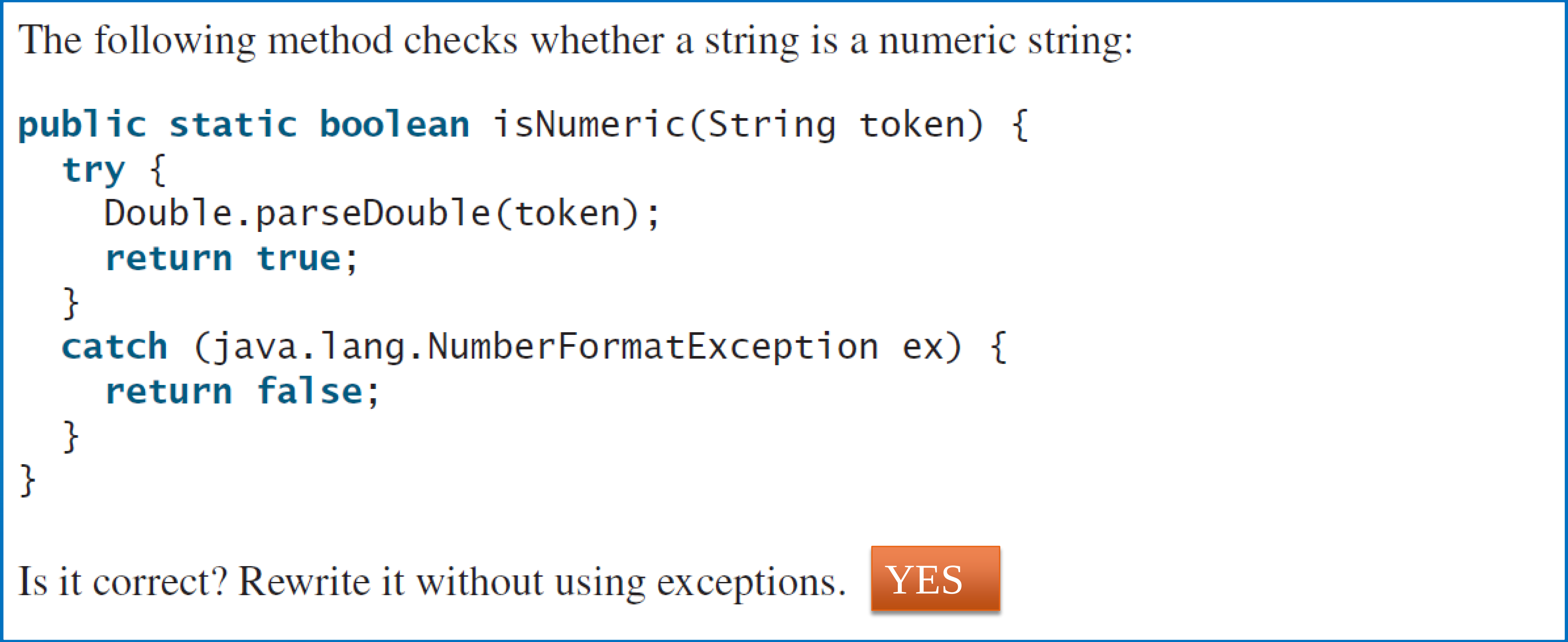
Exercises

What is displayed when the following program is run?

```
public class Test {
  public static void main(String[] args) {
    try {
      int[] list = new int[10];
      System.out.println("list[10] is " + list[10]);
    }
    catch (ArithmeticException ex) {
      System.out.println("ArithmeticException");
    }
    catch (RuntimeException ex) {
      System.out.println("RuntimeException");
    }
    catch (Exception ex) {
      System.out.println("Exception");
    }
  }
}
```

RuntimeException

Exercises



The following method checks whether a string is a numeric string:

```
public static boolean isNumeric(String token) {
  try {
    Double.parseDouble(token);
    return true;
  }
  catch (java.lang.NumberFormatException ex) {
    return false;
  }
}
```

Is it correct? Rewrite it without using exceptions.

YES

Exercises

I have a simple setter method for a property and *null* is not appropriate for this particular property. I have always been torn in this situation: should I throw an *IllegalArgumentException*, or a *NullPointerException*? From the javadocs, both seem appropriate. Is there some kind of an understood standard? Or is this just one of those things that you should do whatever you prefer and both are really correct?

Exercises

Do we throw *null*?

```
class Test {
    public static void main(String args[]){
        try {
            throw null;
        } catch (Exception e){
            System.out.println(e instanceof NullPointerException);
            System.out.println(e instanceof FileNotFoundException);
        }
    }
}
```

Exercises

Can we *return* from a *finally* block as follows:

```
class Test {
    public static void main(String args[]){
        call();
    }
    public static Object call() {
        Object o = null;
        try {   method();   o = new Object(); }
        finally {
            return o;
        }
    }
    public static void method() throws Exception{
        throw new Exception();
    }
}
```

Exercises

Can we catch the exception?

```
class Test {
    public static void main(String args[]) throws Exception{
        call();
    }
    public static Object call() throws Exception{
        Object o = null;
        try { method(); o = new Object(); }
        finally { return o; }
    }
    public static void method() throws Exception{
        throw new Exception();
    }
}
```

Practices

(***ArrayIndexOutOfBoundsException***) Write a program that meets the following requirements:

- Creates an array with **100** randomly chosen integers.
- Prompts the user to enter the index of the array, then displays the corresponding element value. If the specified index is out of bounds, display the message **Out of Bounds**.

(***OutOfMemoryError***) Write a program that causes the JVM to throw an **OutOfMemoryError** and catches and handles this error.