Exception Handling (Chapter 12)

Motivation

```
import java.util.*;

public class ExceptionTest {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        boolean continueInput = true;

        do {
            System.out.print("Enter an integer: ");
            int number = input.nextInt();

            // Display the result
            System.out.println(
              "The number entered is " + number);

            if ( number == -1)
                continueInput = false;

        } while (continueInput);
      }
}
```

Motivation

```
 1  import java.util.*;
 2
 3  public class InputMismatchExceptionDemo {
 4    public static void main(String[] args) {
 5      Scanner input = new Scanner(System.in);
 6      boolean continueInput = true;
 7
 8      do {
 9        try {
10          System.out.print("Enter an integer: ");
11          int number = input.nextInt();
12
13          // Display the result
14          System.out.println(
15            "The number entered is " + number);
16
17          continueInput = false;
18        }
19        catch (InputMismatchException ex) {
20          System.out.println("Try again. (" +
21            "Incorrect input: an integer is required)");
22          input.nextLine(); // discard input
23        }
24      } while (continueInput);
25    }
26  }
```

Exception Terminology

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions
- The following will cause exceptions:
  - Accessing an out-of-bounds array element
  - Writing into a read-only file
  - Trying to read beyond the end of a file
  - Sending illegal arguments to a method
  - Performing illegal arithmetic (e.g divide by 0)
  - Hardware failures

Exception Terminology

- When an exception occurs, we say it was thrown or raised
- When an exception is dealt with, we say it is handled or caught
- The block of code that deals with exceptions is known as an exception handler

Why Use Exceptions?

- Compilation cannot find all errors
- To separate error handling code from regular code
  - Code clarity (debugging, teamwork, etc.)
  - Worry about handling error elsewhere
- To separate error detection, reporting, and handling
- To group and differentiate error types
  - Write error handlers that handle very specific exceptions

Exception-Handling Overview

- *Runtime errors* occur while a program is running, if the JVM detects an operation that is impossible to carry out.
- For example, if you access an array using an index that is out of bounds, you will get a runtime error with an **ArrayIndexOutOfBoundsException**. If you enter a **double** value when your program expects an integer, you will get a runtime error with an **InputMismatchException**.
- In Java, runtime errors are thrown as exceptions.
- 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 will terminate abnormally.

Exception-Handling Overview

- Exception handling enables a program to deal with exceptional situations and continue its normal execution.
- Exceptions can be thrown from a method.
- The caller of the method can catch and handle the exception or chose to let it propagate.

Decoding Exception Messages

```
public class ArrayExceptionExample {
   public static void main(String args[]) {
      String[] names = {“Bilha", “Robert"};
      System.out.println(names[2]);
   }
}
```

The println in the above code causes an exception to be thrown with the following exception message:

Exception in thread "main"

java.lang.ArrayIndexOutOfBoundsException: 2 at

ArrayExceptionExample.main

(ArrayExceptionExample.java:4)

Exception Message Format

- Exception messages have the following format:

  [exception class]: [additional description of exception] at [class].[method] ([file]:[line number]

Exception Messages Example

Exception message from array example

java.lang.ArrayIndexOutOfBoundsException: 2 at
ArrayExceptionExample.main
(ArrayExceptionExample.java:4)

- What is the exception class?

  java.lang.ArrayIndexOutOfBoundsException
- Which array index is out of bounds?

  2
- What method throws the exception?

  ArrayExceptionExample.main
- What file contains the method?

  ArrayExceptionExample.java
- What line of the file throws the exception?

  4

Exception Advantages

Using exception handling enables a method to throw an exception to its caller. Without this capability, a method must handle the exception or terminate the program.

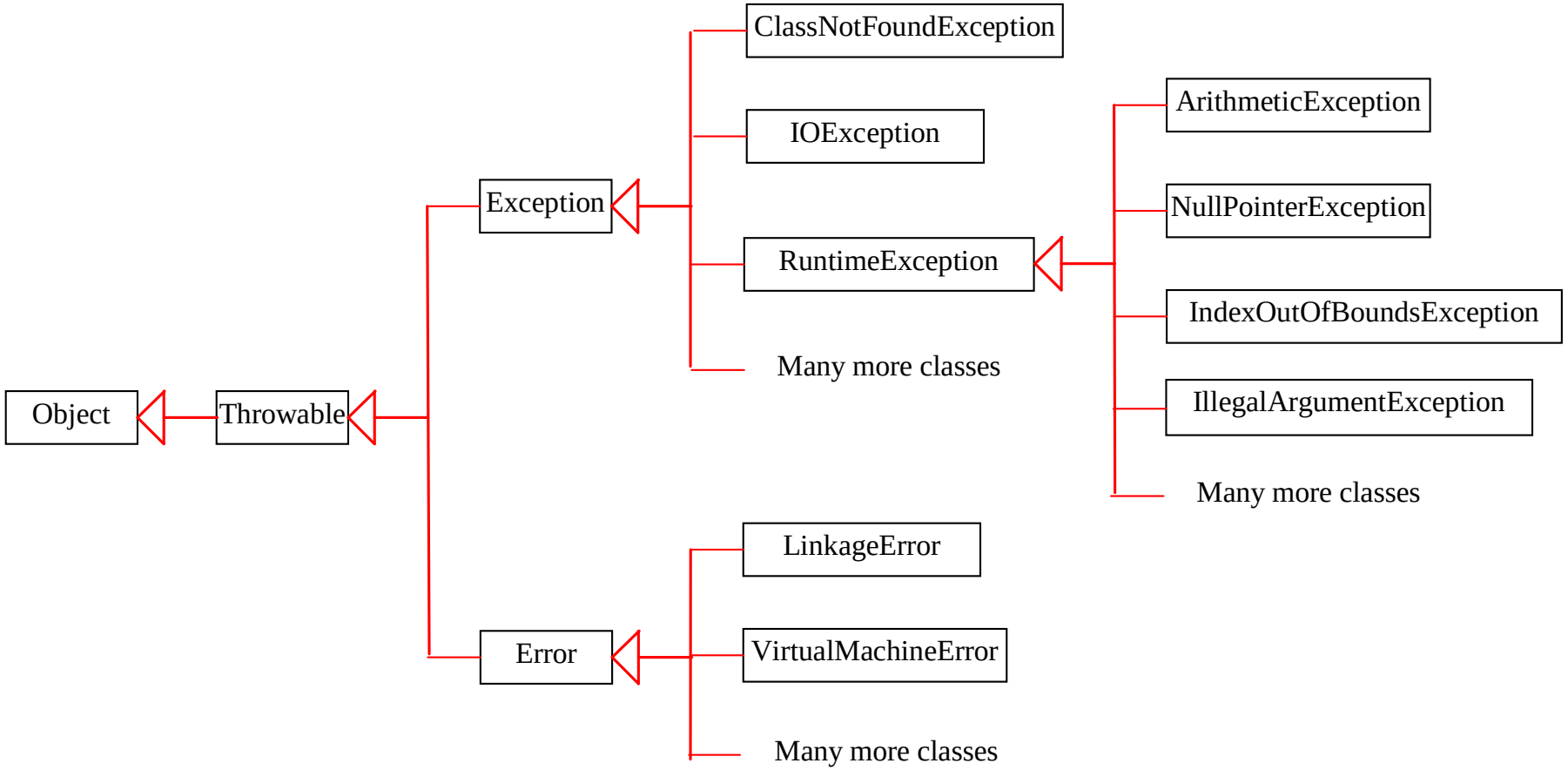
In addition, without exceptions, it is sometimes impossible for a method to deal with unexpected errors.

Handling InputMismatchException

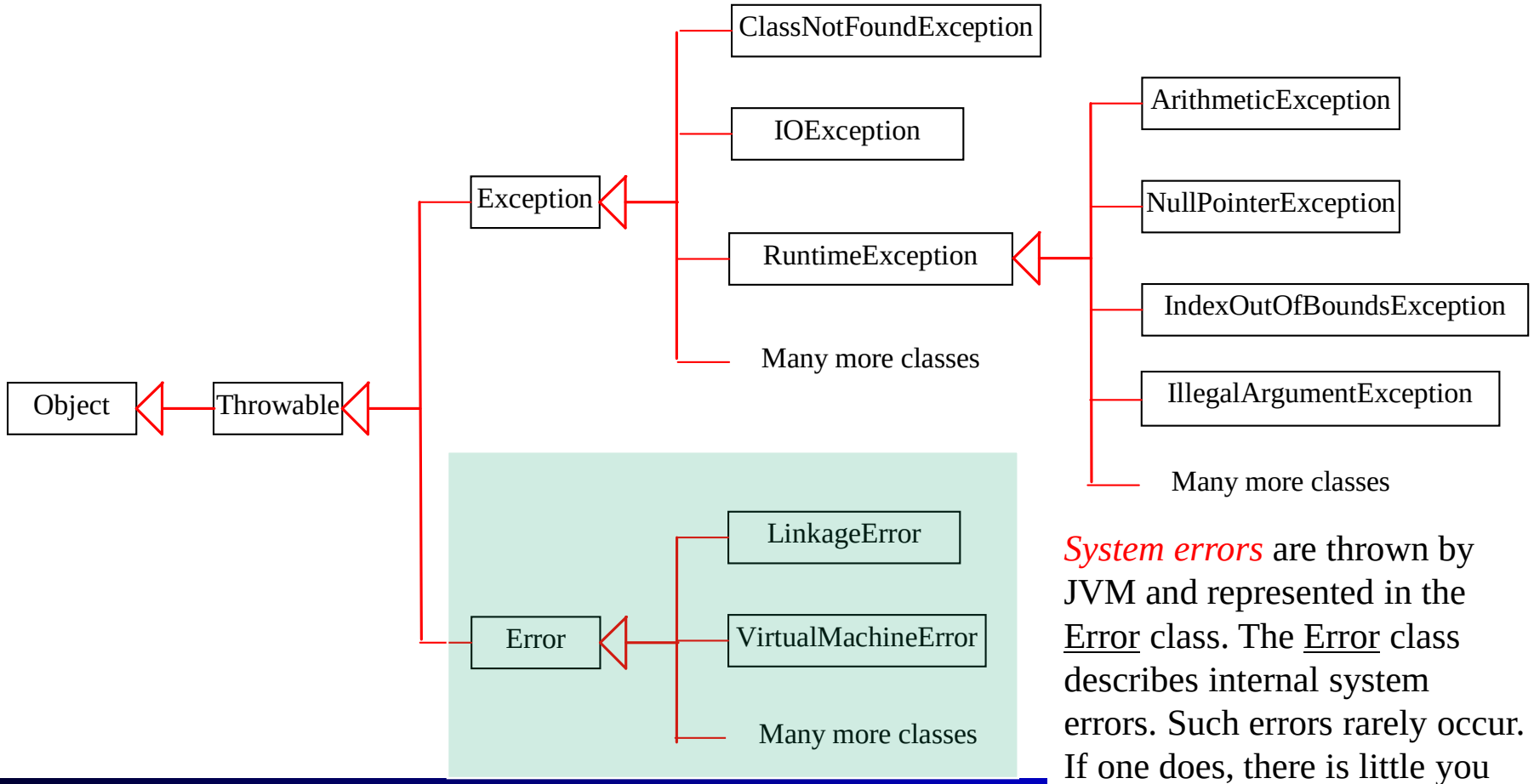
```
 1  import java.util.*;
 2
 3  public class InputMismatchExceptionDemo {
 4    public static void main(String[] args) {
 5      Scanner input = new Scanner(System.in);
 6      boolean continueInput = true;
 7
 8      do {
 9        try {
10          System.out.print("Enter an integer: ");
11          int number = input.nextInt();
12
13          // Display the result
14          System.out.println(
15            "The number entered is " + number);
16
17          continueInput = false;
18        }
19        catch (InputMismatchException ex) {
20          System.out.println("Try again. (" +
21            "Incorrect input: an integer is required)");
22          input.nextLine(); // discard input
23        }
24      } while (continueInput);
25    }
26  }
```

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

Exception Types



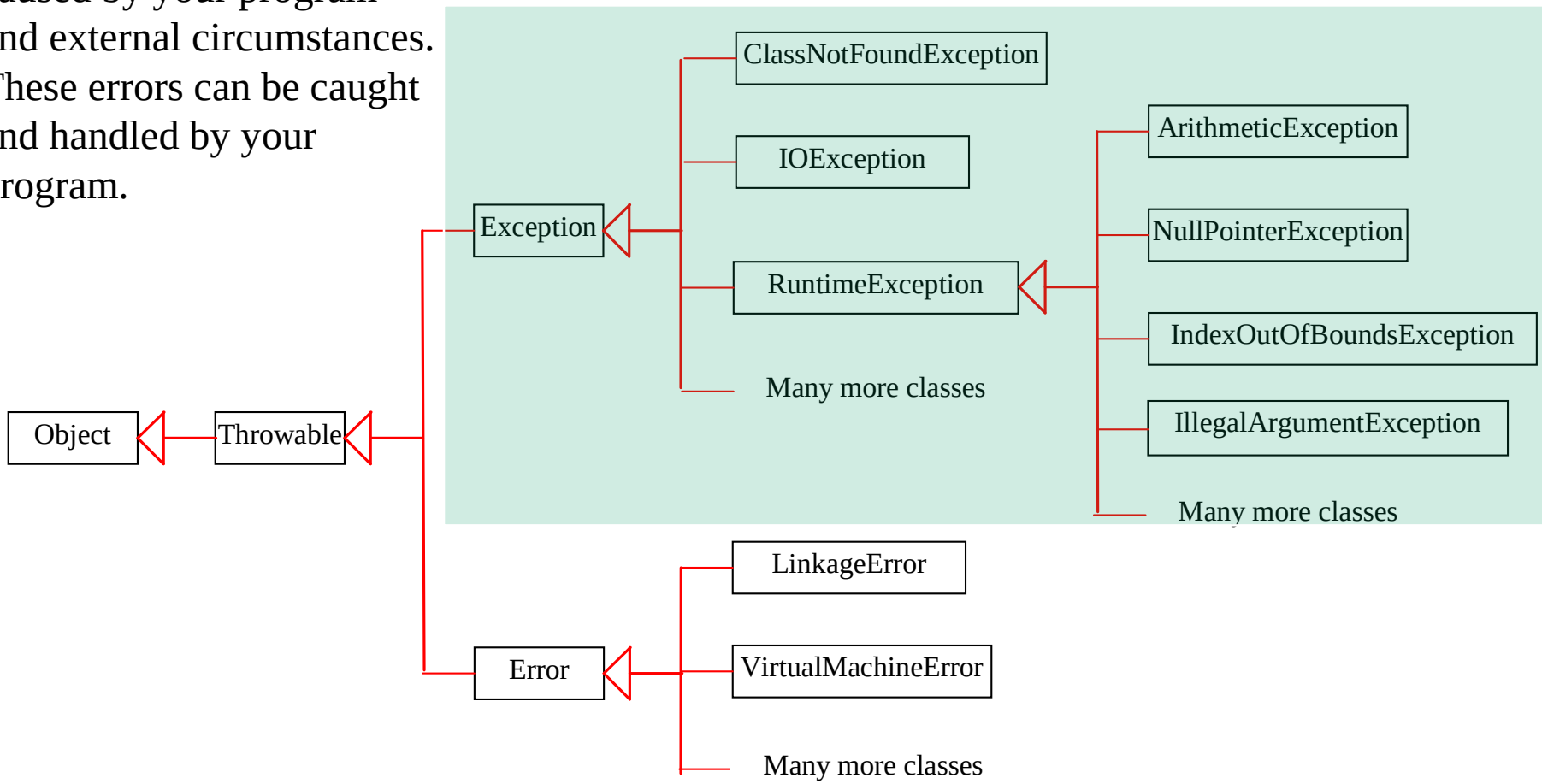
System Errors



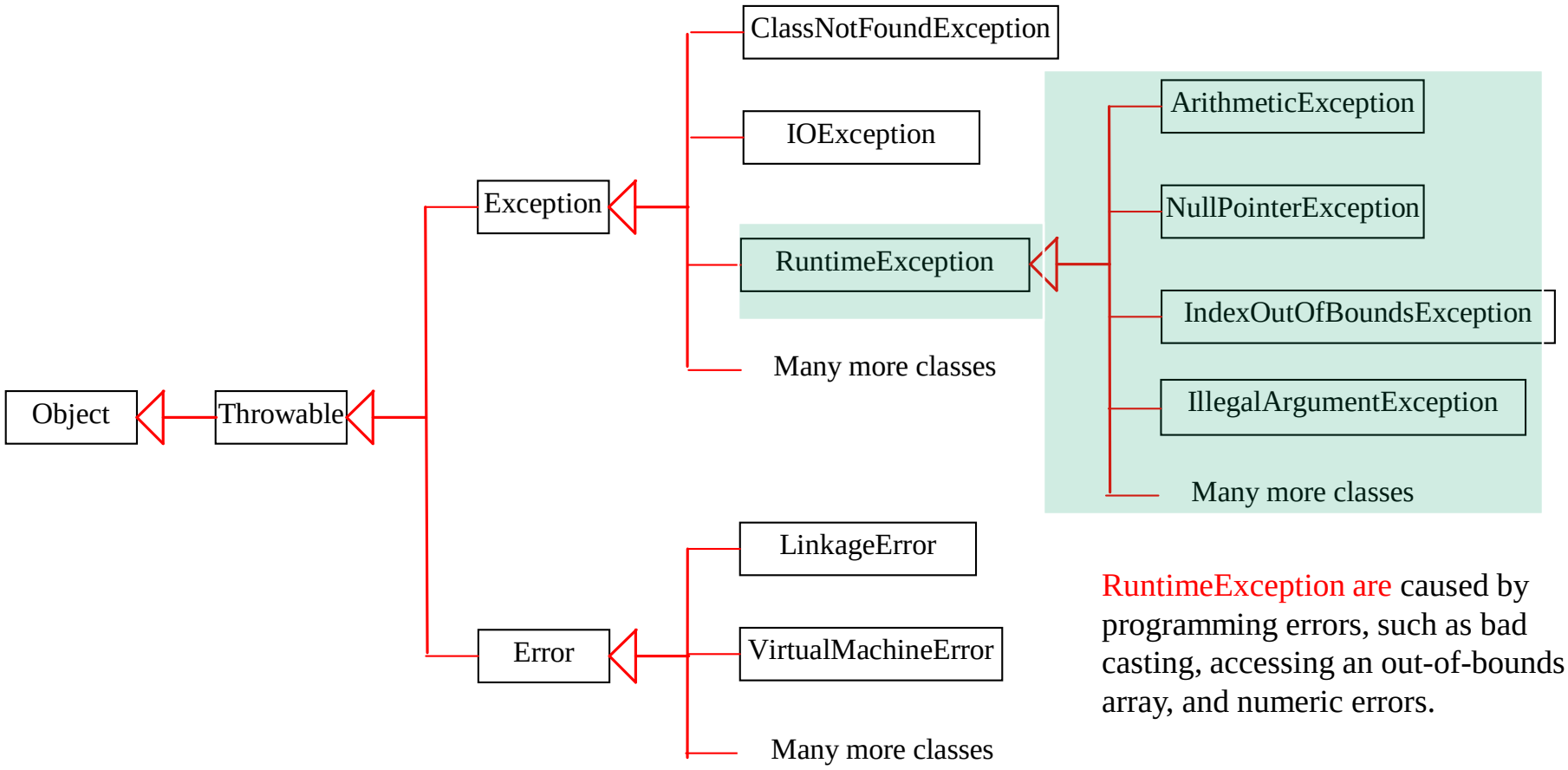
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Exceptions

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



Runtime Exceptions



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

Checked Exceptions vs.Unchecked Exceptions

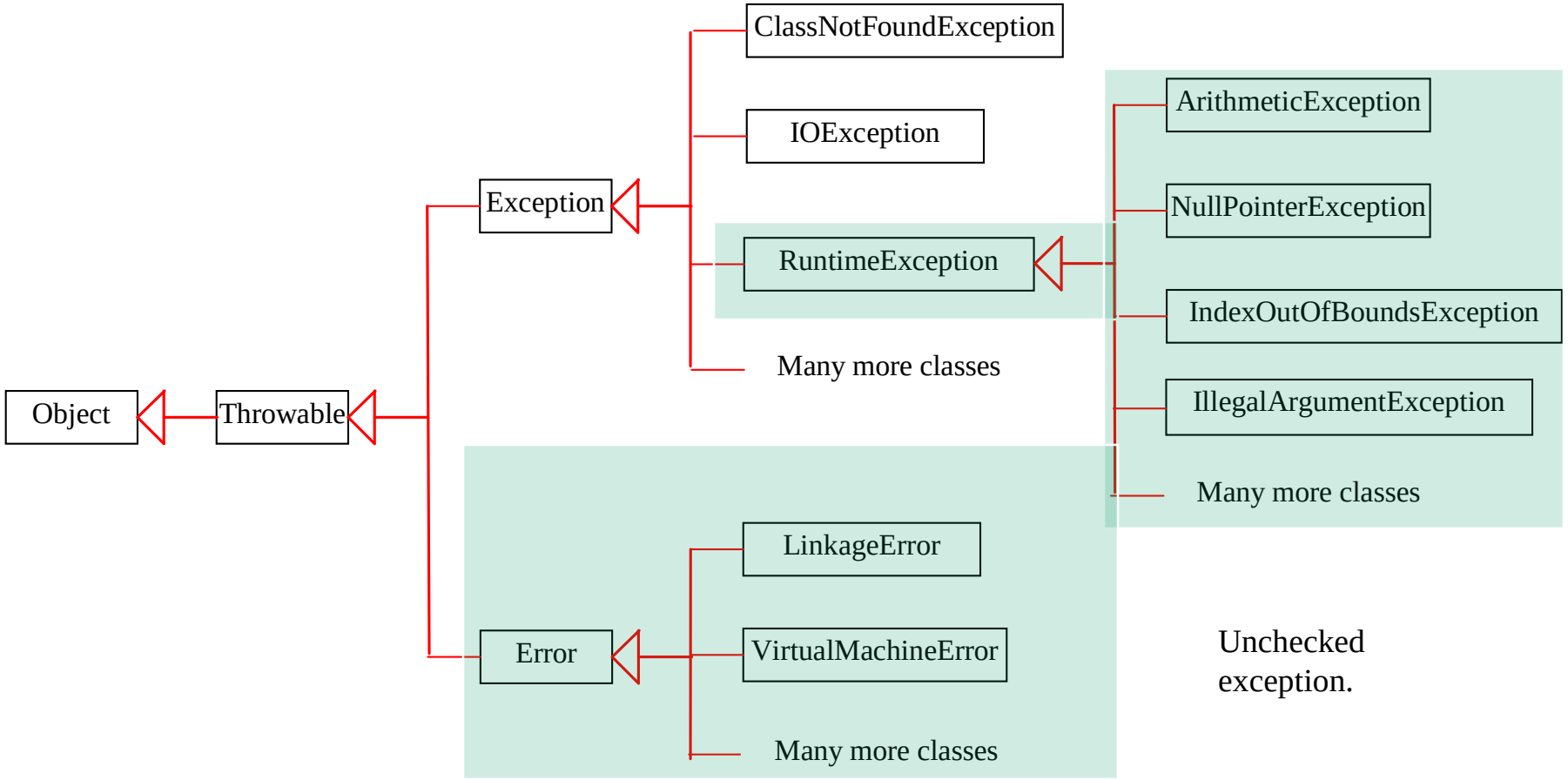
- RuntimeException, Error and their subclasses are known as *unchecked exceptions*.
- All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

Unchecked Exceptions

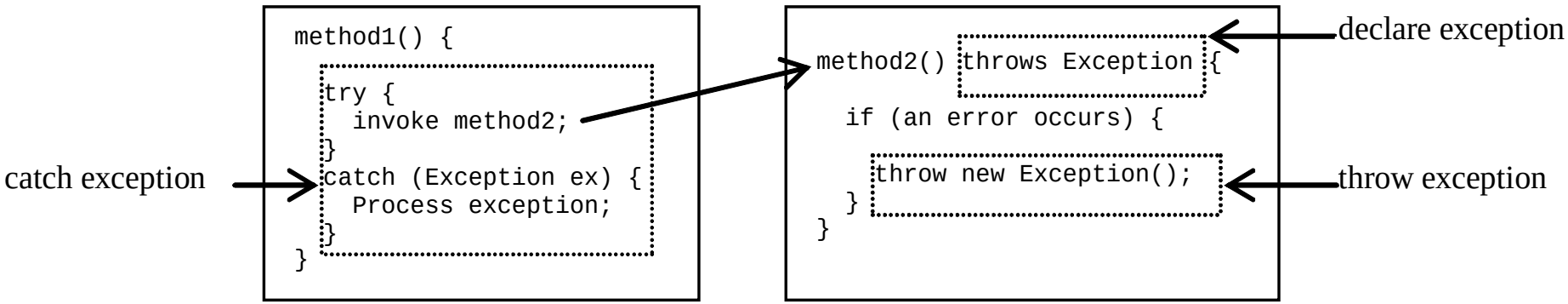
In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. For example, a NullPointerException is thrown if you access an object through a reference variable before an object is assigned to it; an IndexOutOfBoundsException is thrown if you access an element in an array 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



Declaring, Throwing, and Catching Exceptions



Declaring Exceptions

Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.

public void myMethod() throws IOException

public void myMethod() throws IOException, OtherException

Declaring 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");
}
```

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*.
- Use the throw statement to throw an exception
- The throw statement requires a single argument: a Throwable object
  - Throwable objects are instances of any subclass of the Throwable class
  - They include all types of errors and exceptions

Check the API for a full listing of Throwable objects

Throwing Exceptions

Here two examples:

```
if (student == null) {
    NullPointerException ex = new NullPointerException();
    throw ex;
{


if (student == null)
    throw new NullPointerException();
```

Catching Exceptions

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

```
try {
// code that might throw exception
}
catch ([Type of Exception] e) {
// what to do if exception is thrown
}
```

Catching Multiple Exceptions

- Multiple Exceptions can be handled by using multiple successive catch blocks

```
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;
}
```

If no exceptions arise during the execution of the **try** block, the **catch** blocks are skipped.

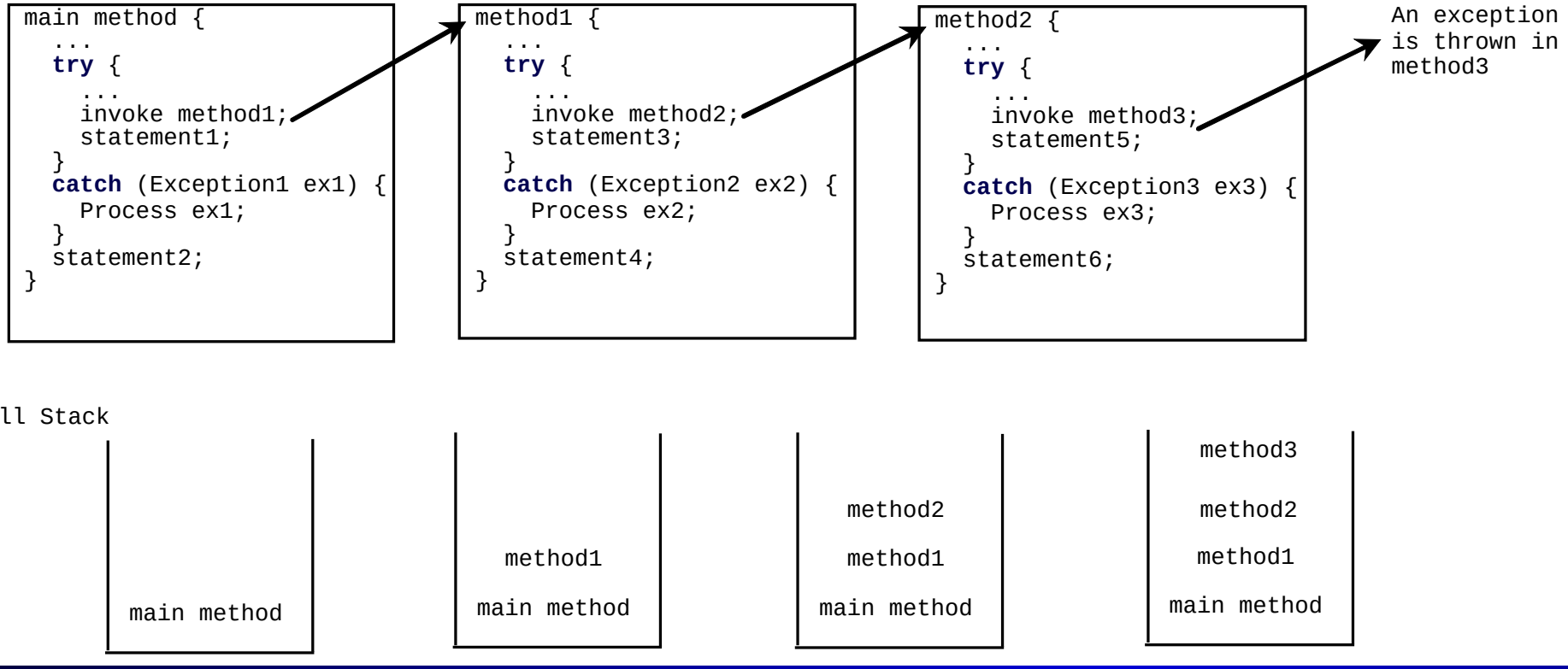
Catching Exceptions

- If one of the statements inside the **try** block throws an exception, Java skips the remaining statements in the **try** block and starts the process of finding the code to handle the exception.
- The code that handles the exception is called the *exception handler*; it is found by *propagating the exception* backward through a chain of method calls, starting from the current method.
- Each **catch** block is examined in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the **catch** block. If so, the exception object is assigned to the variable declared, and the code in the **catch** block is executed.

Catching Exceptions

- If no handler is found, Java exits this method, passes the exception to the method that invoked the method, and continues the same process to find a handler.
- If no handler is found in the chain of methods being invoked, the program terminates and prints an error message on the console.
- The process of finding a handler is called *catching an exception*.

Catching Exceptions



Notes on Catching Exceptions

1. Various exception classes can be derived from a common superclass. If a **catch** block catches exception objects of a superclass, it can catch all the exception objects of the subclasses of that superclass.

2. The order in which exceptions are specified in catch blocks is important. A compile error will result if a catch block for a superclass type appears before a catch block for a subclass type.

Notes on Catching Exceptions cont.

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

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

(a)

```
void p1() throws IOException {

  p2();

}
```

(b)

Example: Declaring, Throwing, and Catching Exceptions

```
public CircleWithException(double newRadius) {
  setRadius(newRadius);
  numberOfObjects++;
}
```

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

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

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

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 re-
 //throwing the exception.
  throw ex;
}
```

The `finally` Block

Occasionally, you may want some code to be executed regardless of whether an exception occurs or is caught. Java has a finally block that can be used to accomplish this objective.

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

The finally block provides a mechanism to clean up regardless of what happens within the try block; for example, it can be used to close files or to release other system resources.

The `finally` Clause

The code in the **finally** block is executed under all circumstances, regardless of whether an exception occurs in the **try** block or is caught. Consider three possible cases:

1. If no exception arises in the **try** block, **finalStatements** is executed, and the next statement after the **try** statement is executed.
2. 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.
3. 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.

The **finally** block executes even if there is a **return** statement prior to reaching the **finally** block.

Notes On 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

- When 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 (provided by Java) 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

```
1  public class InvalidRadiusException extends Exception {
2    private double radius;
3
4    /** Construct an exception */
5    public InvalidRadiusException(double radius) {
6      super("Invalid radius " + radius);
7      this.radius = radius;
8    }
9
10   /** Return the radius */
11   public double getRadius() {
12     return radius;
13   }
14 }
```

Custom Exception Class Example

```
1  public class CircleWithRadiusException {
2    /** The radius of the circle */
3    private double radius;
4
5    /** The number of the objects created */
6    private static int numberOfObjects = 0;
7
8    /** Construct a circle with radius 1 */
9    public CircleWithRadiusException() {
10     this(1.0);
11   }
12
13   /** Construct a circle with a specified radius */
14   public CircleWithRadiusException(double newRadius) {
15     try {
16       setRadius(newRadius);
17       numberOfObjects++;
18     }
19     catch (InvalidRadiusException ex) {
20       ex.printStackTrace();
21     }
22   }
23
24   /** Return radius */
25   public double getRadius() {
26     return radius;
27   }
28
29   /** Set a new radius */
30   public void setRadius(double newRadius)
31       throws InvalidRadiusException {
32     if (newRadius >= 0)
33       radius =  newRadius;
34     else
35       throw new InvalidRadiusException(newRadius);
36   }
37
38   /** Return numberOfObjects */
39   public static int getNumberOfObjects() {
40     return numberOfObjects;
41   }
42
43   /** Return the area of this circle */
44   public double findArea() {
45     return radius * radius * 3.14159;
46   }
47 }
```

Custom Exception Class Example

```
1  public class TestCircleWithRadiusException {
2    /** Main method */
3    public static void main(String[] args) {
4      try {
5        CircleWithRadiusException c1 = new CircleWithRadiusException(5);
6        c1.setRadius(-5);
7        CircleWithRadiusException c3 = new CircleWithRadiusException(0);
8      }
9      catch (InvalidRadiusException ex) {
10       System.out.println(ex);
11     }
12
13     System.out.println("Number of objects created: " +
14       CircleWithRadiusException.getNumberOfObjects());
15   }
16 }
```

Programming Challenge

Implement the hex2Dec(String hexString) method, which converts a hex string into a decimal number. The method throws a NumberFormatException if the string is not a hex string.

Write code to test this method and catch the exception generated by it.