Chapter 12 Exception Handling

Dr. Asem Kitana
Dr. Abdallah Karakra

Motivations: Runtime Errors

- How can you handle the runtime error so that the program can continue to run or terminate gracefully?

Introduction

- *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

Exception Handling

- 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 enables a program to deal with exceptional situations and continue its normal execution.

Runtime Error?

LISTING 14.1 Quotient.java

```
1  import java.util.Scanner;
2
3  public class Quotient {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6
7      // Prompt the user to enter two integers
8      System.out.print("Enter two integers: ");
9      int number1 = input.nextInt();
10     int number2 = input.nextInt();
11
12     System.out.println(number1 + " / " + number2 + " is " +
13       (number1 / number2));
14   }
15 }
```

```
Enter two integers: 5 2 Enter
5 / 2 is 2
```

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

Fix it Using an if Statement

LISTING 14.2 QuotientWithIf.java

```
1  import java.util.Scanner;
2
3  public class QuotientWithIf {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6
7      // Prompt the user to enter two integers
8      System.out.print("Enter two integers: ");
9      int number1 = input.nextInt();
10     int number2 = input.nextInt();
11
12     if (number2 != 0)
13       System.out.println(number1 + " / " + number2
14         + " is " + (number1 / number2));
15     else
16       System.out.println("Divisor cannot be zero ");
17   }
18 }
```

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

Fix it Using try catch

```
import java.lang.ArithmeticException;
import java.util.Scanner;
public class QuotientWithException{
    public static void main(String [] args){
        Scanner input= new Scanner (System.in);
        int number1,number2;
        int result;
        System.out.print("Enter two Integers: ");
        try{
            number1= input.nextInt();
            number2=input.nextInt();
            result= number1/number2;
            System.out.println(number1 + " / " + number2+ " is "+result);
        }
        catch (ArithmeticException ex){
            System.out.println("Exception: an integer cannot be divide by zero" );

        }

        System.out.println("Good Luck!");
    }
}
```

```
Enter two Integers: 2 0
Exception: an integer cannot be divide by zero
Good Luck!
```

LISTING 12.3 QuotientWithMethod.java

```
1  import java.util.Scanner;
2
3  public class QuotientWithMethod {
4    public static int quotient(int number1, int number2) {
5      if (number2 == 0) {
6        System.out.println("Divisor cannot be zero");
7        System.exit(1);
8      }
9
10     return number1 / number2;
11   }
12
13   public static void main(String[] args) {
14     Scanner input = new Scanner(System.in);
15
16     // Prompt the user to enter two integers
17     System.out.print("Enter two integers: ");
18     int number1 = input.nextInt();
19     int number2 = input.nextInt();
20
21     int result = quotient(number1, number2);
22     System.out.println(number1 + " / " + number2 + " is "
23       + result);
24   }
25 }
```

```
Enter two integers: 5 3 ↵Enter
5 / 3 is 1
```

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

LISTING 14.4 QuotientWithException.java

```
1  import java.util.Scanner;
2
3  public class QuotientWithException {
4    public static int quotient(int number1, int number2) {
5      if (number2 == 0)
6        throw new ArithmeticException("Divisor cannot be zero");
7
8      return number1 / number2;
9    }
10
11   public static void main(String[] args) {
12     Scanner input = new Scanner(System.in);
13
14     // Prompt the user to enter two integers
15     System.out.print("Enter two integers: ");
16     int number1 = input.nextInt();
17     int number2 = input.nextInt();
18
19     try {
20       int result = quotient(number1, number2);
21       System.out.println(number1 + " / " + number2 + " is "
22         + result);
23     }
24     catch (ArithmeticException ex) {
25       System.out.println("Exception: an integer " +
26         "cannot be divided by zero ");
27     }
28
29     System.out.println("Execution continues ...");
30   }
31 }
```

throw exception

If an ArithmeticException occurs

```
Enter two integers: 5 3 Enter
5 / 3 is 1
Execution continues ...
```

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

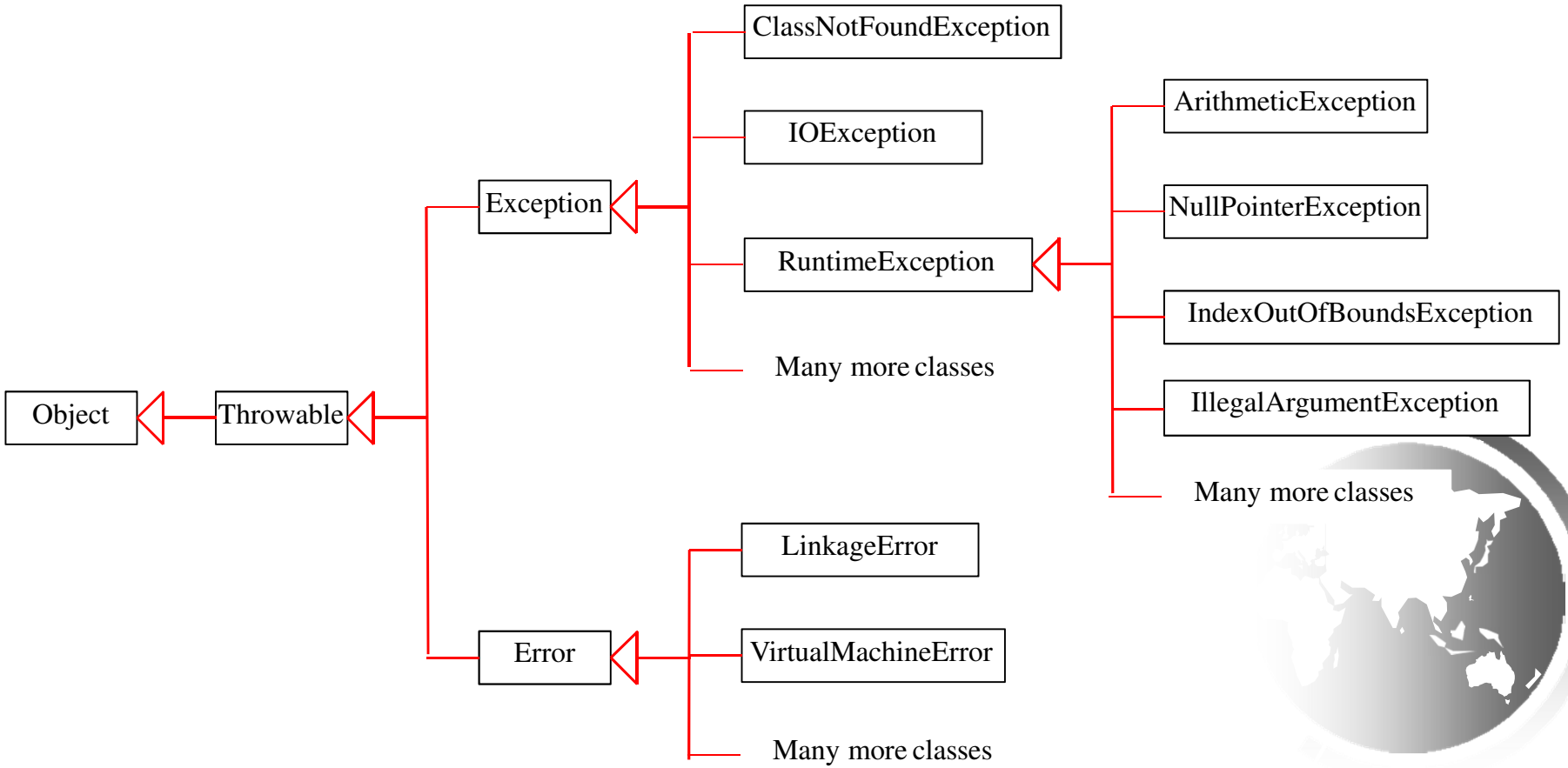
Throw statement

The value thrown, in this case **new ArithmeticException("Divisor cannot be zero"),** is called an *exception*. The execution of a **throw** statement is called *throwing an exception*.

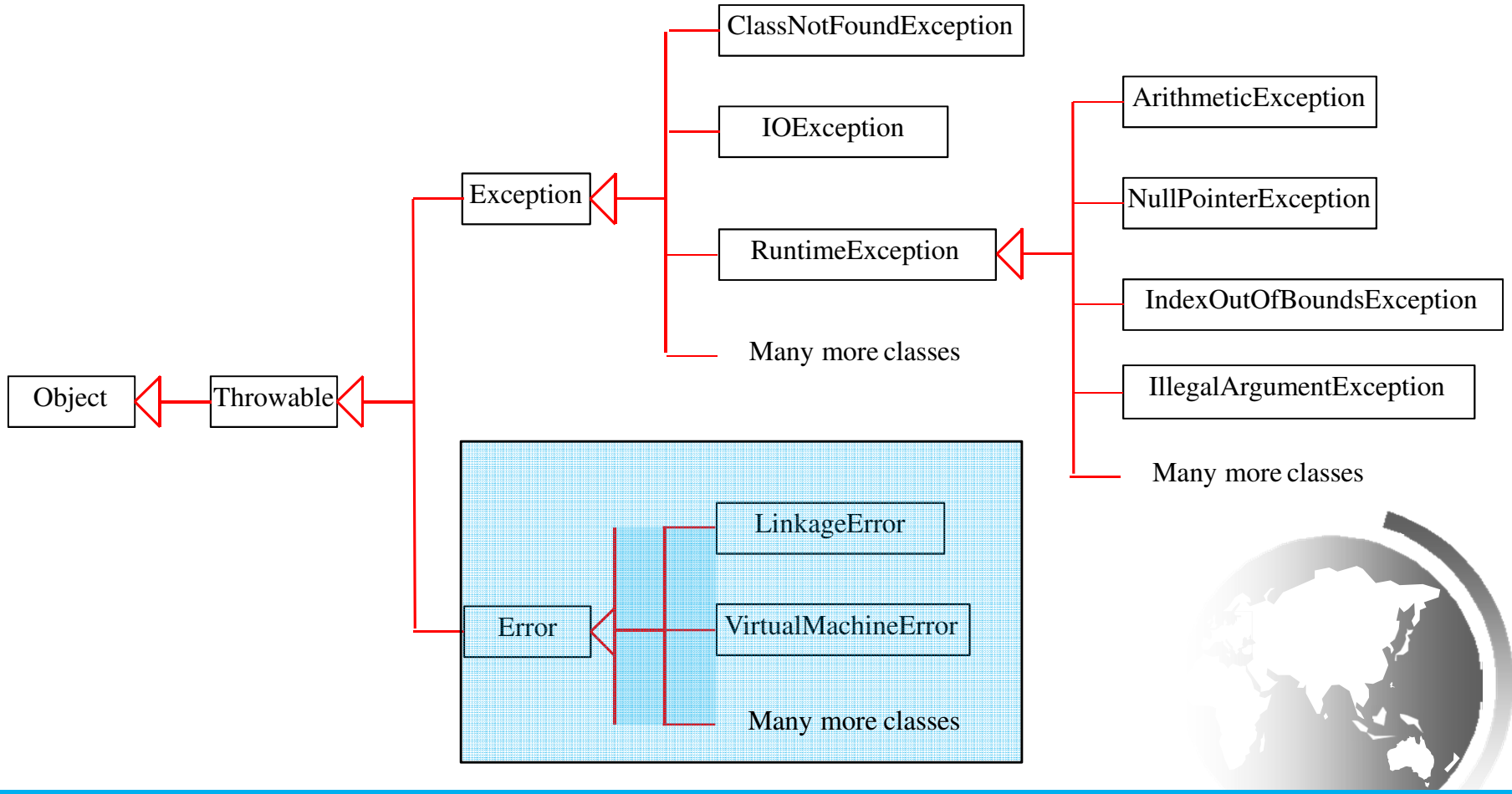
The exception is an object created from an exception class. In this case, the exception class is **java.lang.ArithmeticException**. The constructor **ArithmeticException(str)** is invoked to construct an exception object, where **str** is a message that describes the exception.

Exception Types

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

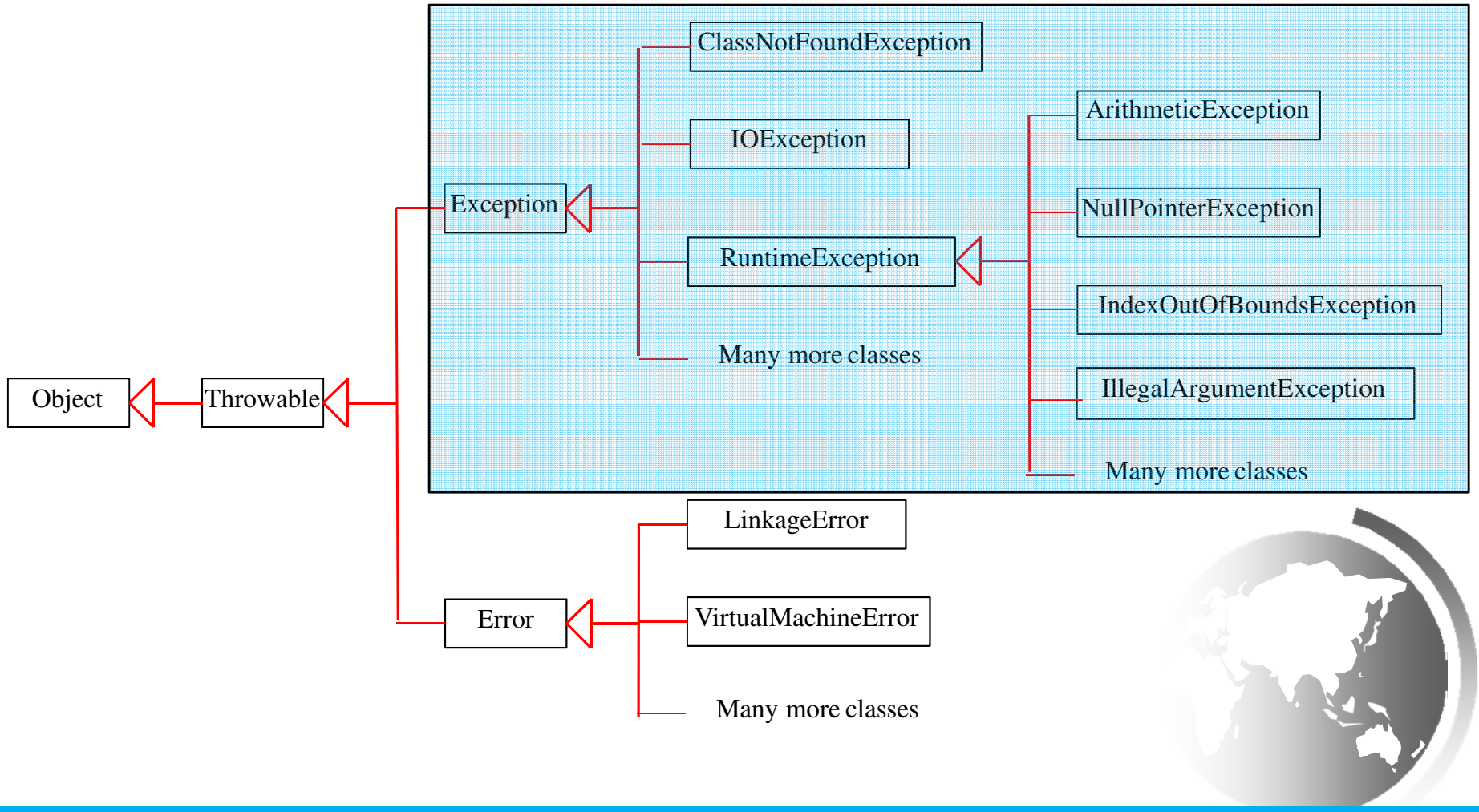


System Errors



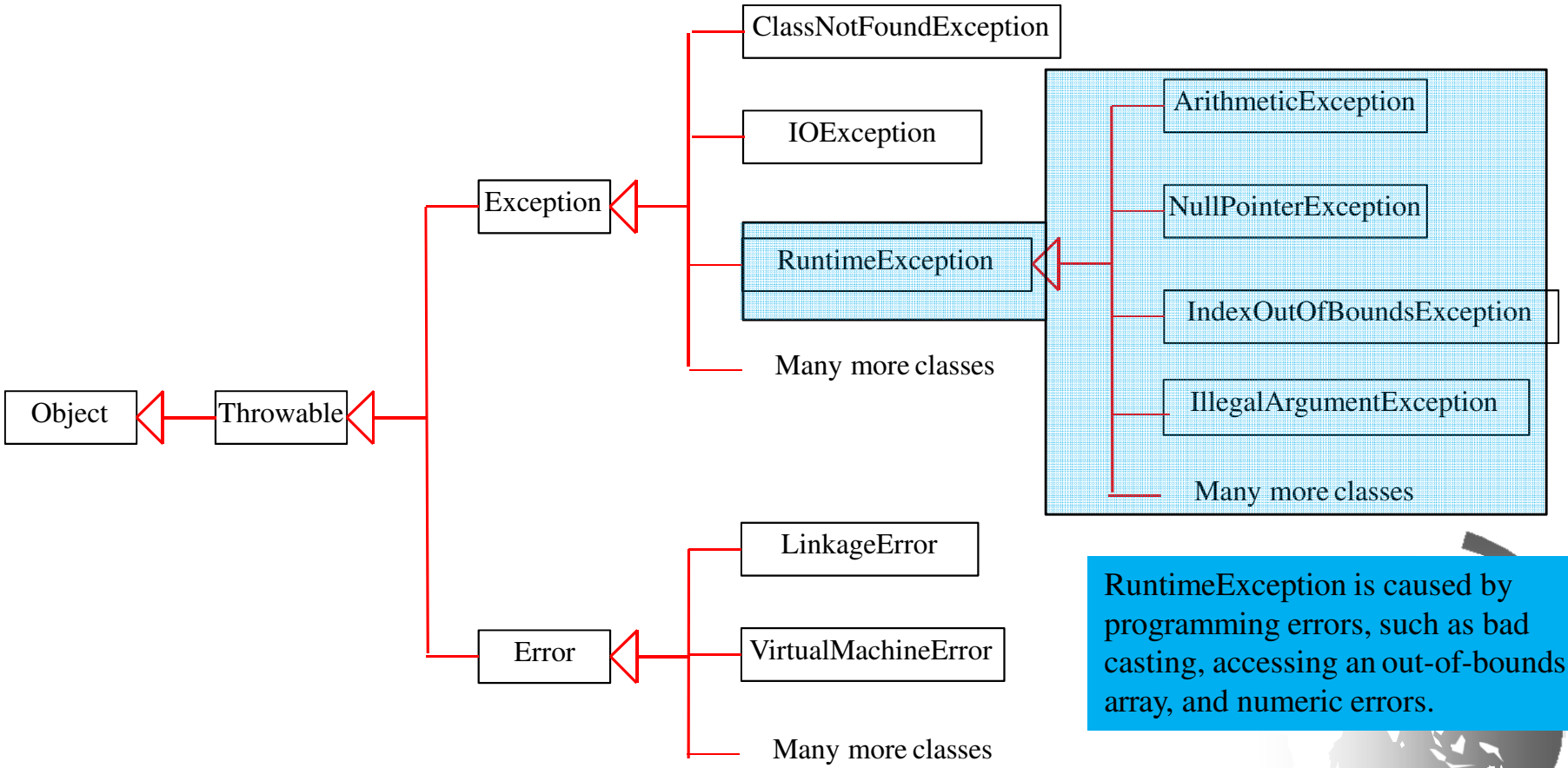
System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

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.

Exception

TABLE 12.2 Examples of Subclasses of `Exception`

Class	*Reasons for Exception*
`ClassNotFoundException`	Attempt to use a class that does not exist. This exception would occur, for example, if you tried to run a nonexistent class using the **java** command, or if your program were composed of, say, three class files, only two of which could be found.
`IOException`	Related to input/output operations, such as invalid input, reading past the end of a file, and opening a nonexistent file. Examples of subclasses of `IOException` are `InterruptedIOException`, `EOFException` (EOF is short for End of File), and `FileNotFoundException`.

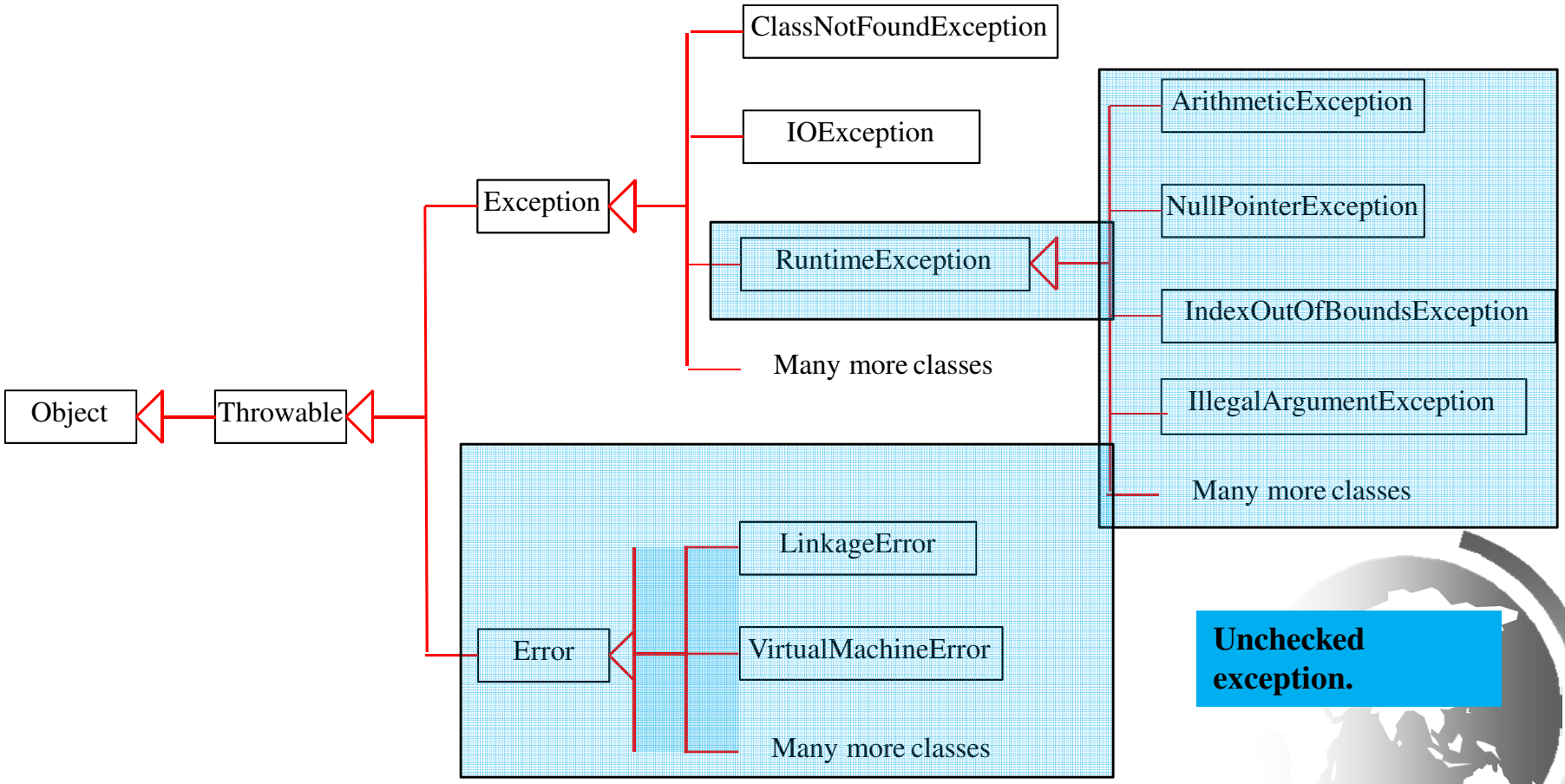
TABLE 12.3 Examples of Subclasses of `RuntimeException`

Class	*Reasons for Exception*
`ArithmeticException`	Dividing an integer by zero. Note that floating-point arithmetic does not throw exceptions (see Appendix E, Special Floating-Point Values).
`NullPointerException`	Attempt to access an object through a `null` reference variable.
`IndexOutOfBoundsException`	Index to an array is out of range.
`IllegalArgumentException`	A method is passed an argument that is illegal or inappropriate.

Checked Exceptions vs. Unchecked Exceptions

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

Unchecked Exceptions



Unchecked exception.

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.

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

Note

If a method does not declare exceptions in the superclass, you cannot override it to declare exceptions in subclas

Handling Exceptions

Java forces you to deal with checked exceptions.

Two possible ways to deal:

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

(a)

```
void p1() throws IOException {

  riskyMethod();

}
```

(b)

Checked exceptions are checked at compile-time. It means if the method is throwing a checked exception, then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error. It is named as checked exception because these exceptions are checked at Compile time.

We can resolve the checked exception by two ways.

1. Declare the exception using throws keyword.
2. Handle them using try-catch blocks

checked exceptions

```
public static void WriteToFile (String s) throws FileNotFoundException{
    File file = new File ("testFile2");
    PrintWriter printFile= new PrintWriter(file);

     printFile.println(s);
     printFile.close();
}
```

Declare the exception using throws keyword

checked exceptions

```
public static void WriteToFile (String s){
      File file;
      PrintWriter printFile=null;
   try{
         file = new File ("testFile2.txt");
         printFile= new PrintWriter(file);

         printFile.println(s);
      }
   catch (FileNotFoundException e){
         System.out.println(".......");
      }
         printFile.close();

}
```

Handle the exception using try-catch blocks

Checked Exceptions Example

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.File;

public class FileExample {


 public static void WriteToFile (String s) throws FileNotFoundException{
    File file = new File ("testFile2");
    PrintWriter printFile= new PrintWriter(file);

     printFile.println(s);
     printFile.close();
 }
    public static void main(String[] args) {

        try {
          WriteToFile("Hello Comp 231");
      } catch (FileNotFoundException e) {

          System.out.println(e.toString());
      }
    }
}
```

For Your Information

When somebody writes a code that could encounter a **runtime error**,

- it creates an object of appropriate Exception class and throws it
- and must also declare it in case of checked exception

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

Tip

The keyword to declare an exception is **throws**, and the keyword to throw an exception

Catching Exceptions

- Install an exception handler with **try/ catch** statement

```
try {
   //Statements that may throw exceptions
}

catch (Exception1 exVar1) {
  //code to handle exceptions of type Exception1;
}

catch (Exception2 exVar2) {
  // code to handle exceptions of type Exception2;
}

...
catch (ExceptionN exVarN) {
  // code to handle exceptions of type exceptionN;
}

// statement after try-catch block
```