Exception Handling

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
- To explore the advantages of using exception handling
- To distinguish exception types: Error (fatal) vs. Exception (nonfatal)
- To declare exceptions in a method header
- To throw exceptions in a method
- To write a try-catch block to handle exceptions
- To explain how an exception is propagated
- To obtain information from an exception object

Introduction

- *Runtime errors* occur while a program is running if the JVM detects an operation that is impossible to carry out.
- 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

- Quotient.java

```java
import java.util.Scanner;
public class Quotient {

    public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    //Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    System.out.println(number1 + "/" + number2 + " is "+ (number1/number2));
    }
}
```

RUN

Exception-Handling Overview

- QuotientWithIf.java:

```
import java.util.Scanner;
public class QuotientWithIf {

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);

        //Prompt the user to enter two integers
        System.out.print("Enter two integers: ");
        int number1 = input.nextInt();
        int number2 = input.nextInt();

        if (number2 !=0)
        System.out.println(number1 + "/" + number2 + " is + (number1/number2));
        else
        System.out.println("Divisor cannot be  zero ");
    }
}
```

RUN

Exception-Handling Overview

- QuotienWithMethod.java:

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

RUN

Exception Advantages

- You should **not** let the method terminate the program—the *caller* should decide whether to terminate the program.
- How can a method notify its caller an exception has occurred?
- Java enables a method to throw an exception that can be caught and handled by the caller.

```java
import java.util.Scanner;

public class QuotientWithException {

    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 ...");
    }

    public static int quotient(int number1, int number2) {
        if (number2 == 0)
            throw new ArithmeticException("Divisor cannot be zero");
        return number1/number2;
    }
}
```

RUN

Exception Advantages

- If **number2** is **0**, the method throws an exception by executing

```
throw new ArithmeticException("Divisor cannot be zero");
```

- The constructor **ArithmeticException(str)** is invoked to construct an exception object, where **str** is a message that describes the exception.
- When an exception is thrown, the normal execution flow is interrupted.
- The statement for invoking the method is contained in a **try** block. The **try** block contains the code that is executed in normal circumstances.
- The exception is caught by the **catch** block. The code in the **catch** block is executed to *handle the exception.*
- 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.

Example: InputMismatchExceptionDemo

```
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);
    }
}
```

RUN

Try/Catch Statement

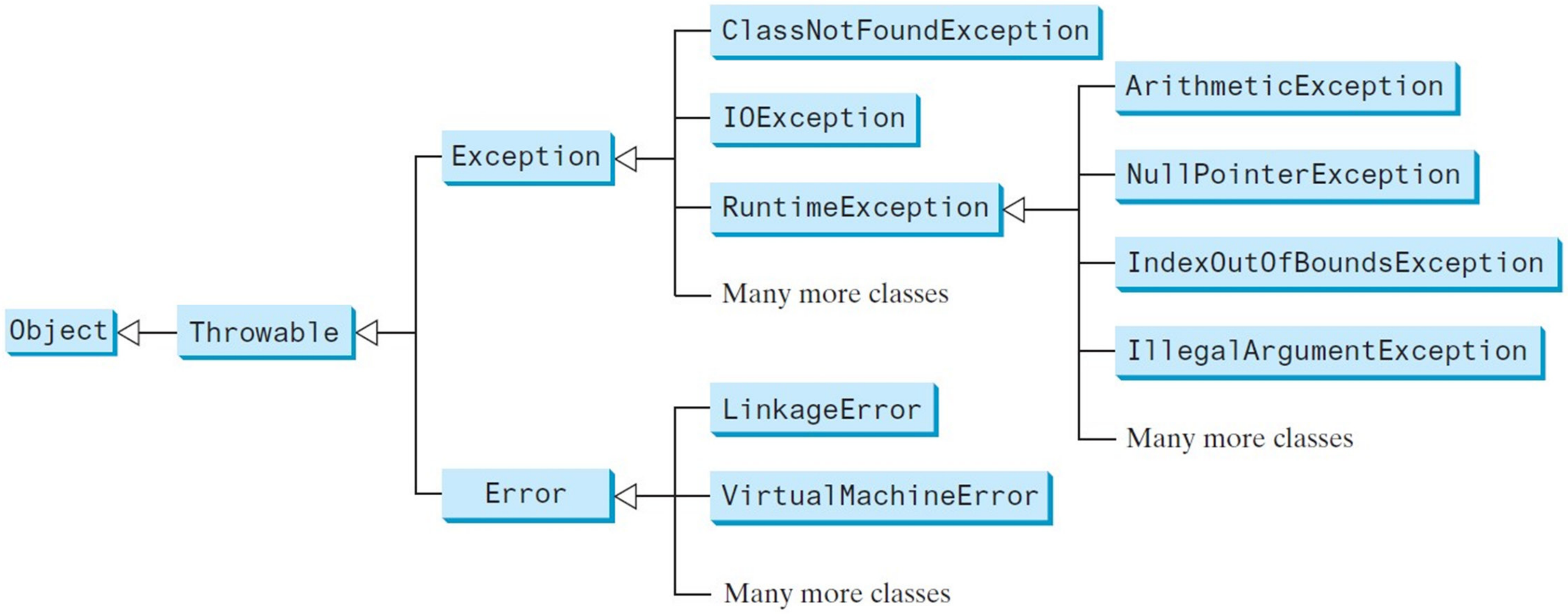
- A *try/catch* statement consists of *try* block and one or more *catch* blocks.
- A *try* block contains the statements that might throw exceptions.
- When a statement in a *try* block throws an exceptions, the rest of the statements in the *try* block are skipped and control is transferred to the *catch* block.

```
try {
        ....

} catch (Exception name) {
        ....

} catch (Exception name) {
        ....
}
```

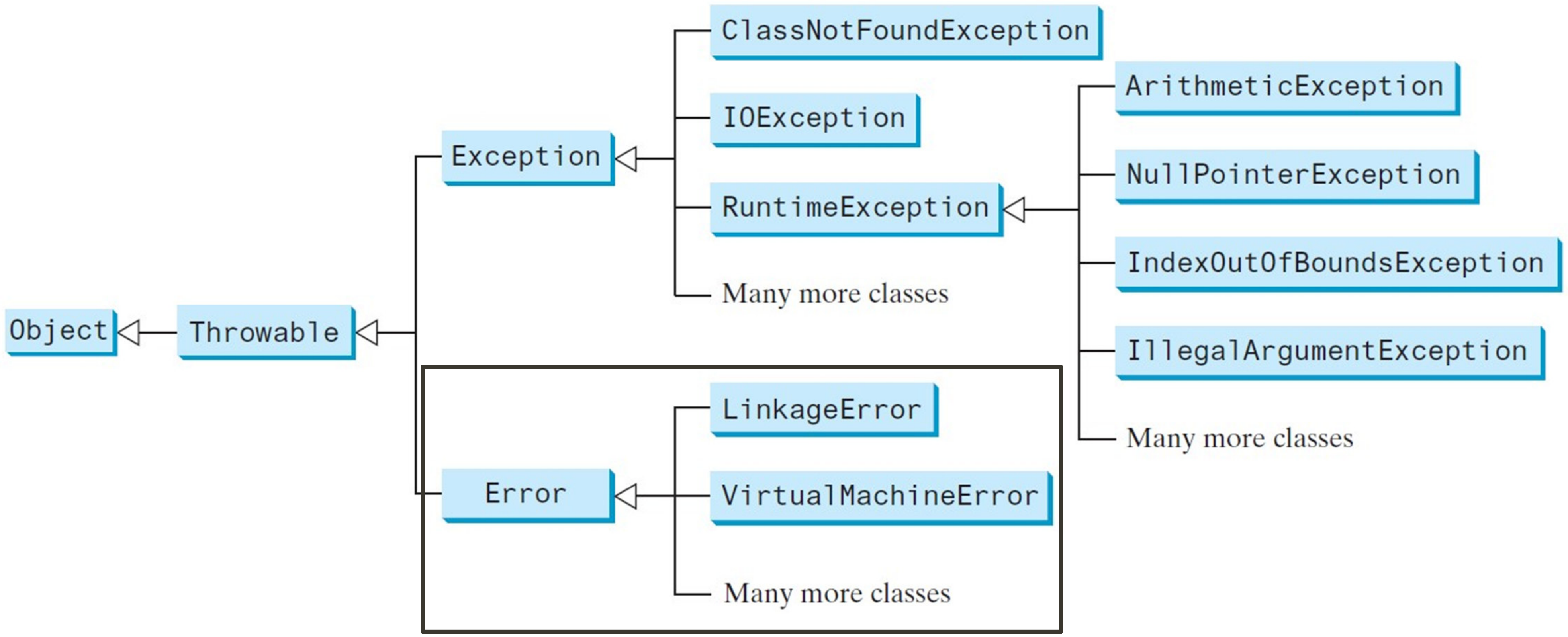
Exception Types



Exception Types

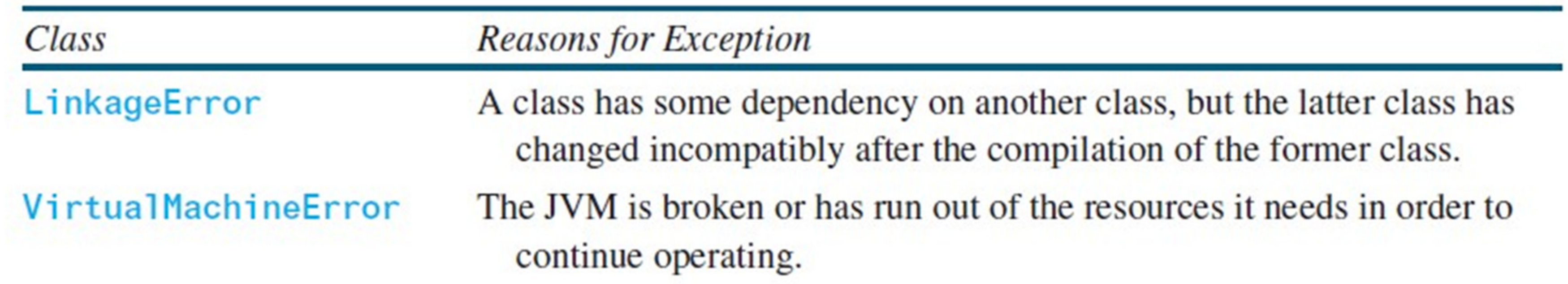
- The **Throwable** class is the root of exception classes.
- You can create your own exception classes by extending **Exception** or a subclass of **Exception**.
- The exception classes can be classified into three major types: *system errors*, *exceptions*, and *runtime exceptions*.

System Errors



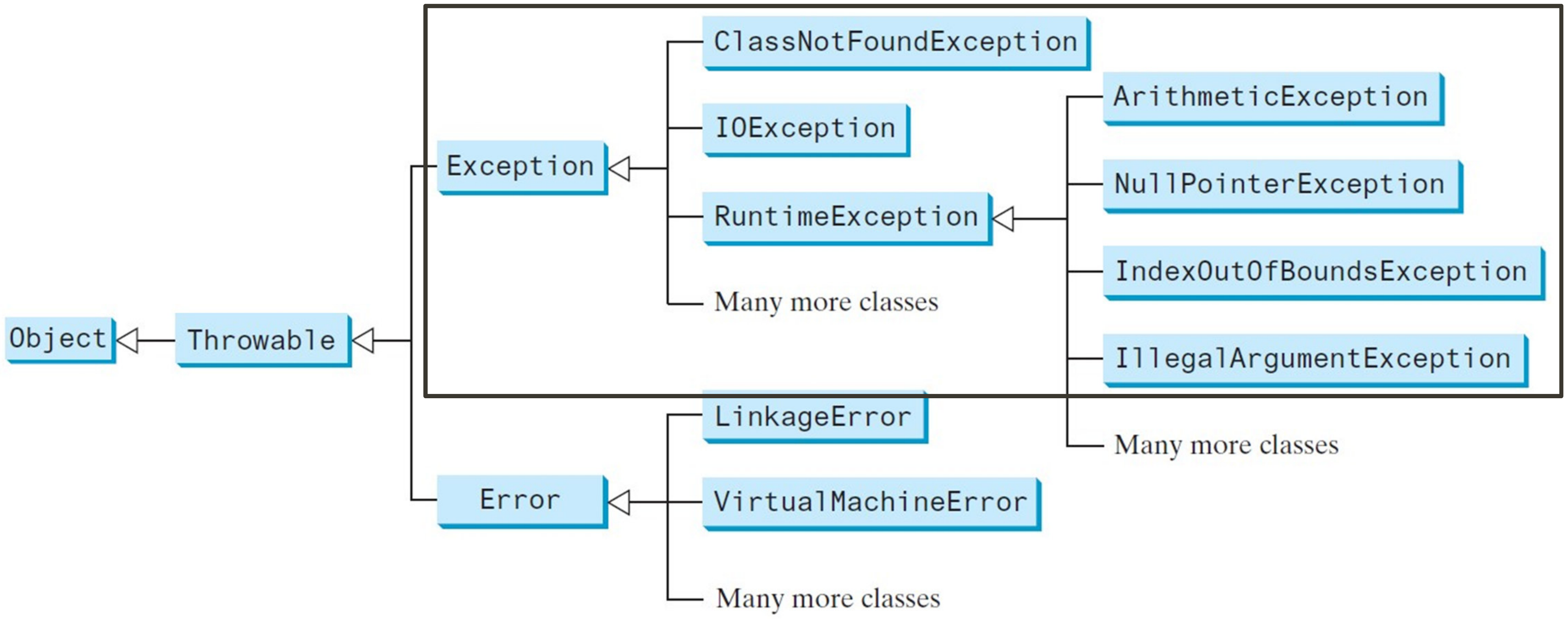
System Errors

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



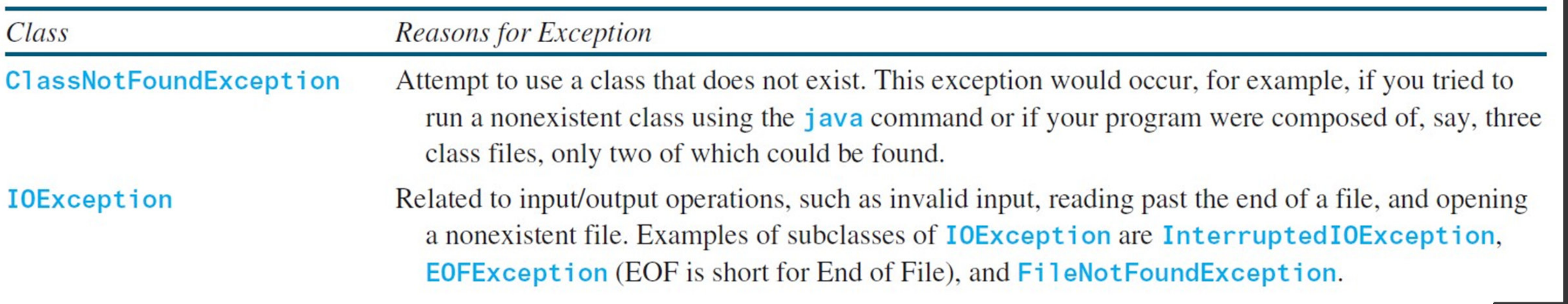
Class	*Reasons for Exception*
`LinkageError`	A class has some dependency on another class, but the latter class has changed incompatibly after the compilation of the former class.
`VirtualMachineError`	The JVM is broken or has run out of the resources it needs in order to continue operating.

Exceptions


Object
Throwable
Exception
ClassNotFoundException
IOException
RuntimeException
Many more classes
ArithmeticException
NullPointerException
IndexOutOfBoundsException
IllegalArgumentException
Many more classes
Error
LinkageError
VirtualMachineError
Many more classes

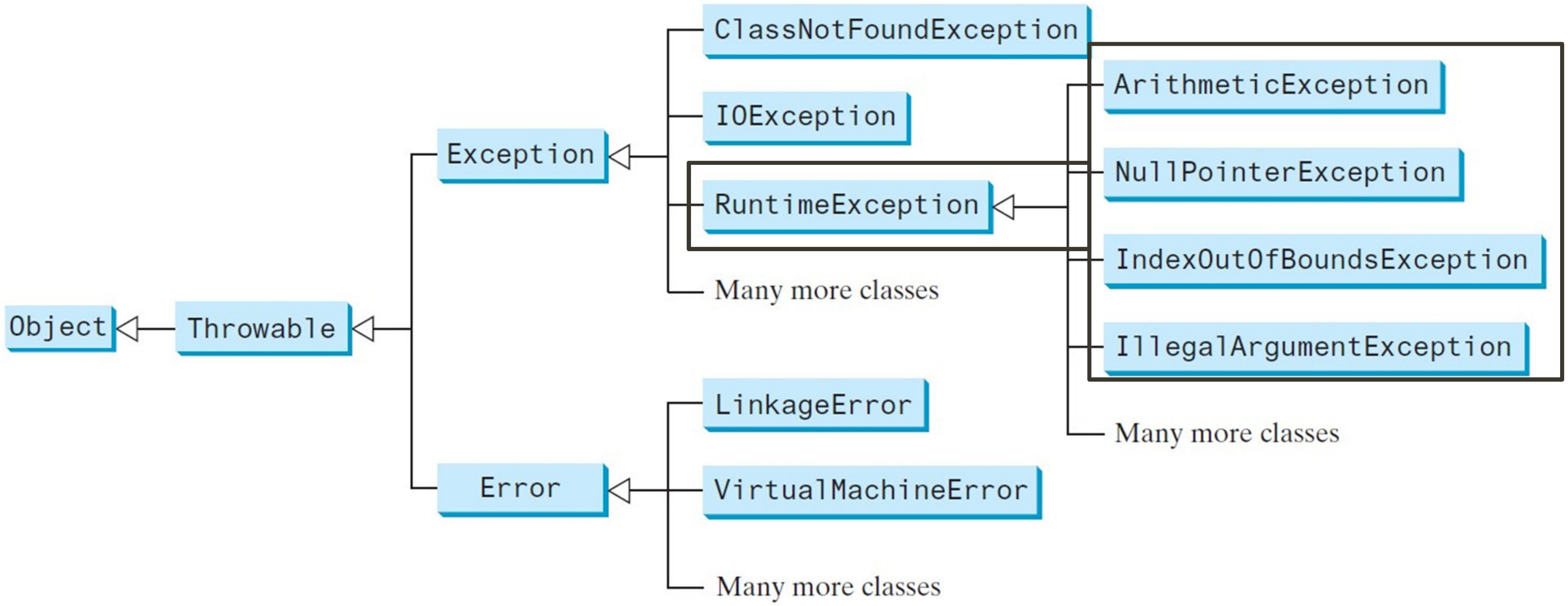
Exceptions

- *Exceptions* are represented in the **Exception** class, which describes errors caused by your program and by external circumstances.
- These errors can be caught and handled by your program.
- **Subclasses of Exception Class:**



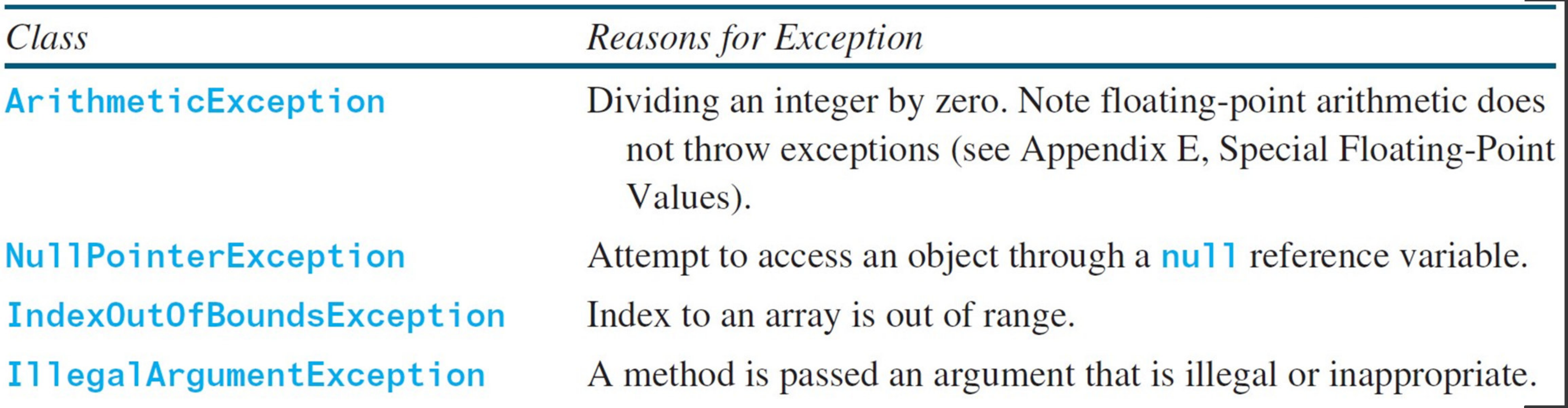
Class	*Reasons for Exception*
`ClassNotFoundException`	Attempt to use a class that does not exist. This exception would occur, for example, if you tried to run a nonexistent class using the `java` command or if your program were composed of, say, three class files, only two of which could be found.
`IOException`	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`.

Runtime Exceptions



Runtime Exceptions

- *Runtime exceptions* are represented in the **RuntimeException** class, which describes programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors. Runtime exceptions normally indicate programming errors.
- Subclasses of RuntimeExcetpions



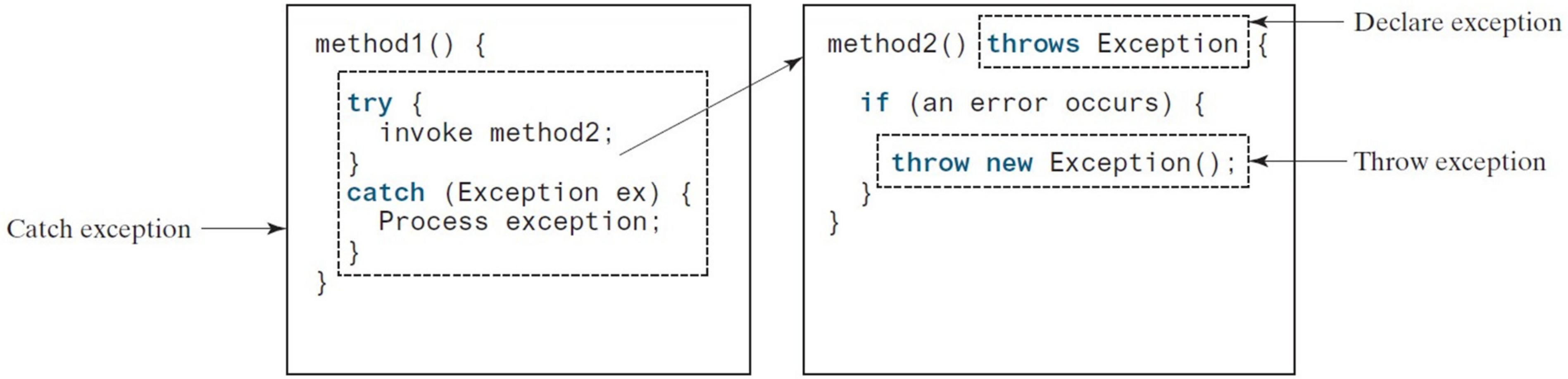
Class	*Reasons for Exception*
`ArithmeticException`	Dividing an integer by zero. Note floating-point arithmetic does not throw exceptions (see Appendix E, Special Floating-Point Values).
`NullPointerException`	Attempt to access an object through a `null` reference variable.
`IndexOutOfBoundsException`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

Checked and Unchecked Exceptions

- **RuntimeException**, **Error**, and their subclasses are known as *unchecked exceptions*.
- All other exceptions are known as *checked exceptions*, meaning the compiler forces the programmer to check and deal with them in a **try-catch** block or declare it in the method header.
- In most cases, unchecked exceptions reflect programming logic errors that are unrecoverable.
- To avoid cumbersome overuse of **try-catch** blocks, Java does not mandate that you write code to catch or declare unchecked exceptions.

Declaring, Throwing, and Catching Exceptions

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



```
method1() {
  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {
  if (an error occurs) {
    throw new Exception();
  }
}
```

Declaring Exceptions

- In Java, every method must state the types of *checked exceptions* it might throw. This is known as *declaring exceptions*.
- Because system errors and runtime errors can happen to any code, Java does not require that you declare **Error** and **RuntimeException** (unchecked exceptions) explicitly in the method.
- However, all other exceptions thrown by the method must be explicitly declared in the method header so the caller of the method is informed of the exception.

Declaring Exceptions

- To declare an exception in a method, use the **throws** keyword in the method header, as in this example:

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

- If the method might throw multiple exceptions, add a list of the exceptions, separated by commas, after **throws**:

```
public void myMethod()
   throws Exception1, Exception2, ..., ExceptionN
```

Throwing Exceptions

- A program that detects an error can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*.
- Here is an example: Suppose the program detects that an argument passed to the method violates the method contract; the program can create an instance of **IllegalArgumentException** and throw it, as follows:

```
IllegalArgumentException ex =new IllegalArgumentException("Wrong Argument");
throw ex;
```

- Or, if you prefer, you can use the following:

```
throw new IllegalArgumentException("Wrong Argument");
```

- **Tip**: The keyword to declare an exception is **throws**, and the keyword to throw an exception is **throw**.

Throwing Exceptions Example

```java
public void setRadius (double newRadius) throws IllegalArgumentException{

    if (newRadius > 0) {
        radius = newRadius;
    }else {
        throw new IllegalArgumentException("Radius cannot be  negative");
    }

}
```

Catching Exceptions

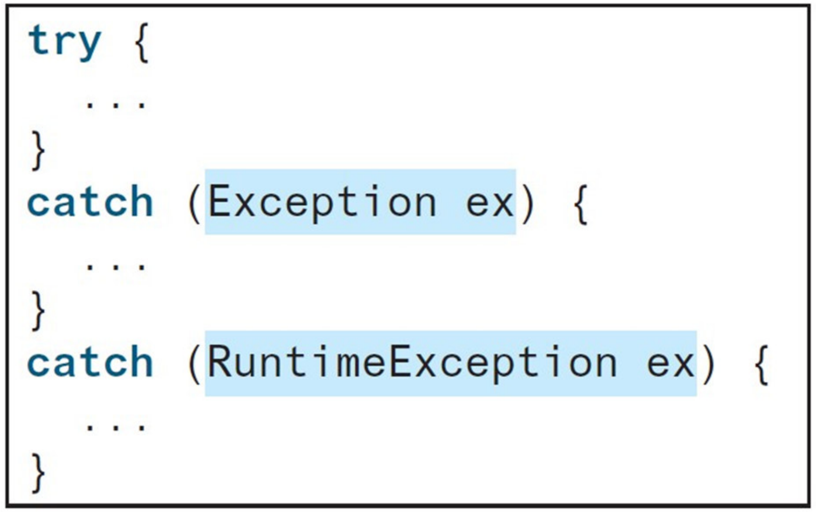
- When an exception is thrown, it can be caught and handled in a **try-catch** block, as follows:

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

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

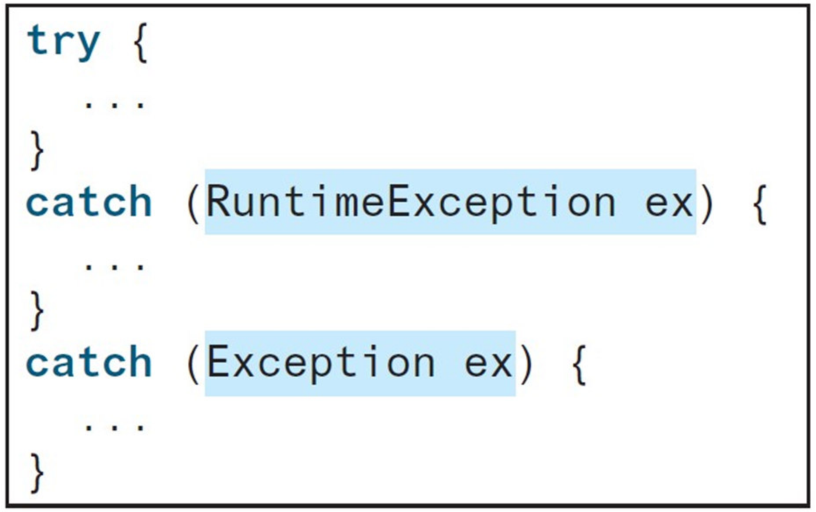
Catching Exceptions

- **Note:** 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. For example, the ordering in (a) below is erroneous, because **RuntimeException** is a subclass of **Exception**. The correct ordering should be as shown in (b).



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

(a) Wrong order



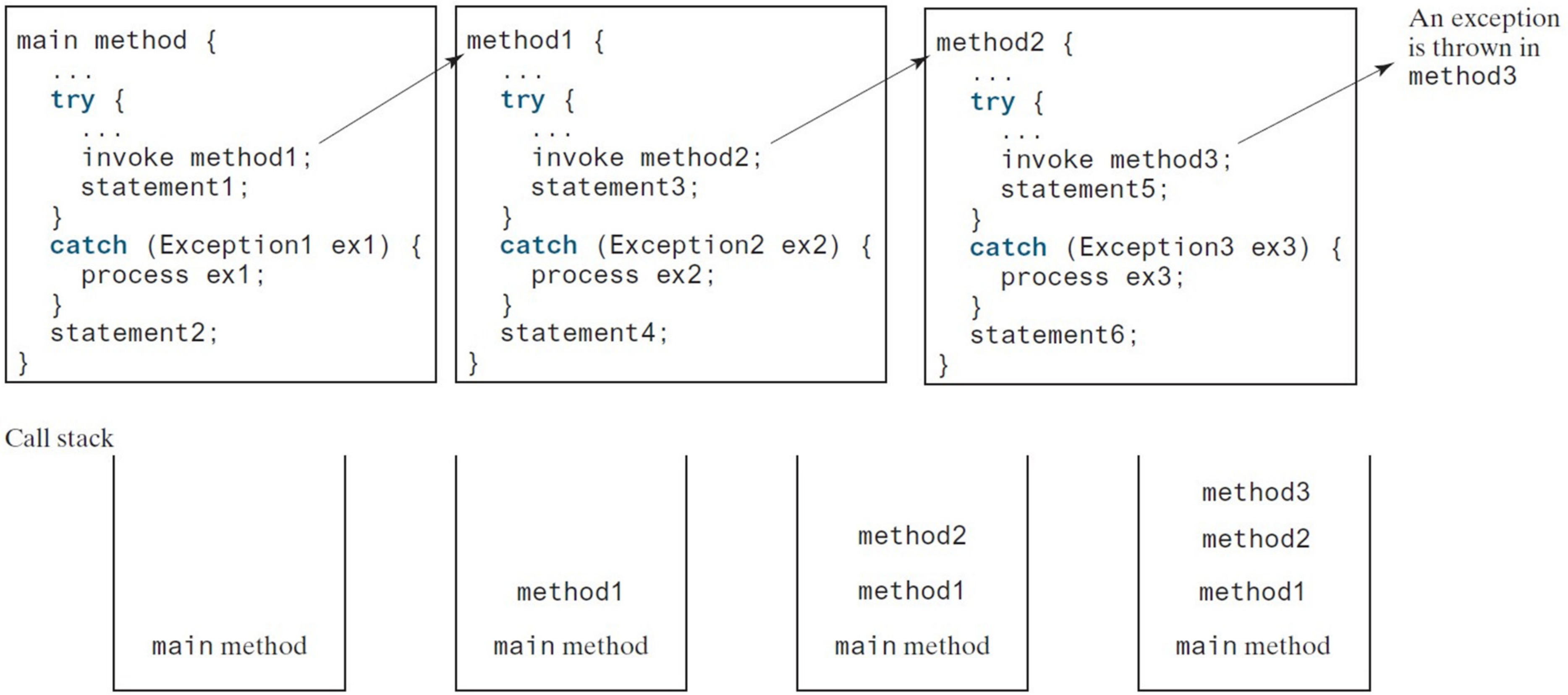
```
try {
  ...
}
catch (RuntimeException ex) {
  ...
}
catch (Exception ex) {
  ...
}
```

(b) Correct order

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. If no handler is found, Java exits this method, passes the exception to the method's caller, 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.

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;
}
```

Example: Catching Exceptions

```java
public class CircleWithException {
    /** The radius of the circle */
    private double radius;

    /** The number of the objects created */
    private static int numberOfObjects = 0;

    /** Construct a circle with radius 1 */
    public CircleWithException() {
        this(1.0);
    }

    /** Construct a circle with a specified radius */
    public CircleWithException(double newRadius) {
        setRadius(newRadius);
        numberOfObjects++;
    }

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

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

    /** Return numberOfObjects */
    public static int getNumberOfObjects() {
        return numberOfObjects;
    }

    /** Return the area of this circle */
    public double findArea() {
        return radius * radius * 3.14159;
    }
}
```

Example: Catching Exceptions

```
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());

    }

}
```

RUN

The finally Clause

- Occasionally, you may want some code to be executed regardless of whether an exception occurs or is caught.
- Java has a **finally** clause that can be used to accomplish this objective. The syntax for the **finally** clause might look like this:

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

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.

Cautions: When to Use Exceptions

- The **try** block contains the code that is executed in normal circumstances. The **catch** block contains the code that is executed in exceptional circumstances. 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 exception through the chain of method calls to search for the handler.

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 or use exceptions.

When to Use Exceptions

- When should you use a **try-catch** block in the code?
- Use it when you have to deal with unexpected error conditions. Do not use a **try-catch** block 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:

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