COMP30820
Java Programming (Conv)

Michael O’Mahony

Chapter 12 Exception Handling and Text I/O

Motivations

A program will terminate when a runtime error is encountered.

In Java, runtime errors are thrown as *exceptions*.

For example, if you access an array using an index that is out of bounds, you will get a runtime error: `ArrayIndexOutOfBoundsException`.

If the exception is not handled, the program will terminate abnormally.

How can you handle the exception so that the program can continue to run or else terminate gracefully?

Objectives

- To get an overview of exceptions and exception handling (§12.2).
- 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 obtain information from an exception object (§12.4.4).
- To use the `finally` clause in a `try-catch` block (§12.5).
- To discover file/directory properties using the `File` class (§12.10).
- To write data to a file using the `PrintWriter` and `FileWriter` classes (§12.11.1).
- To read data from a file using the `Scanner` class (§12.11.3).

Example #1 – Integer Division by Zero

```
public class Divide1 {

   public static int divide(int n1, int n2) {
        return n1 / n2;
   }

   public static void main(String[] args) {
        int x = 1;
        int y = 0;

        int result = divide(x, y);
        System.out.println("result: " + result);
   }
}
```

Example #1 – Integer Division by Zero

```
public class Divide1 {

   public static int divide(int n1, int n2) {
        return n1 / n2;
   }

   public static void main(String[] args) {
        int x = 1;
        int y = 0;

        int result = divide(x, y);
        System.out.println("result: " + result);
   }
}
```

A runtime error (`ArithmeticException`) occurs, because an integer cannot be divided by 0.

The program is terminated which is clearly a problem…

Example #2 – Integer Division by Zero

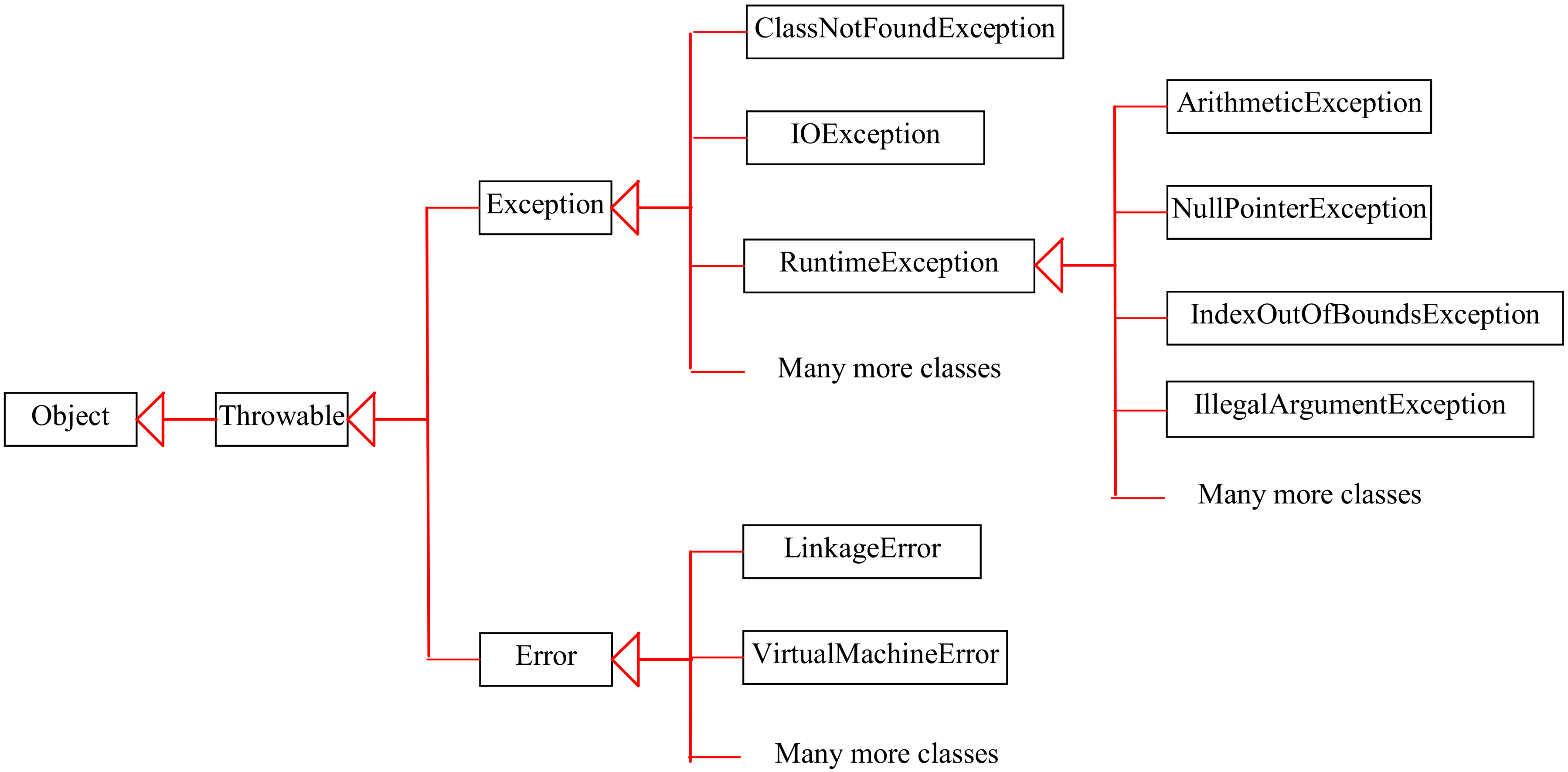
```
public class Divide2 {

   public static int divide(int n1, int n2) throws ArithmeticException {
        if (n2 == 0)
           throw new ArithmeticException("Divisor cannot be zero");

        return n1 / n2;
   }

   public static void main(String[] args) {
        int x = 1;
        int y = 0;

        try {
           int result = divide(x, y);
           System.out.println("result: " + result);
        }
        catch (ArithmeticException ex) {
           System.out.println("Exception: integer division by zero");
        }

        System.out.println("Execution continues ...");
   }
}
```

Example #2 – Integer Division by Zero

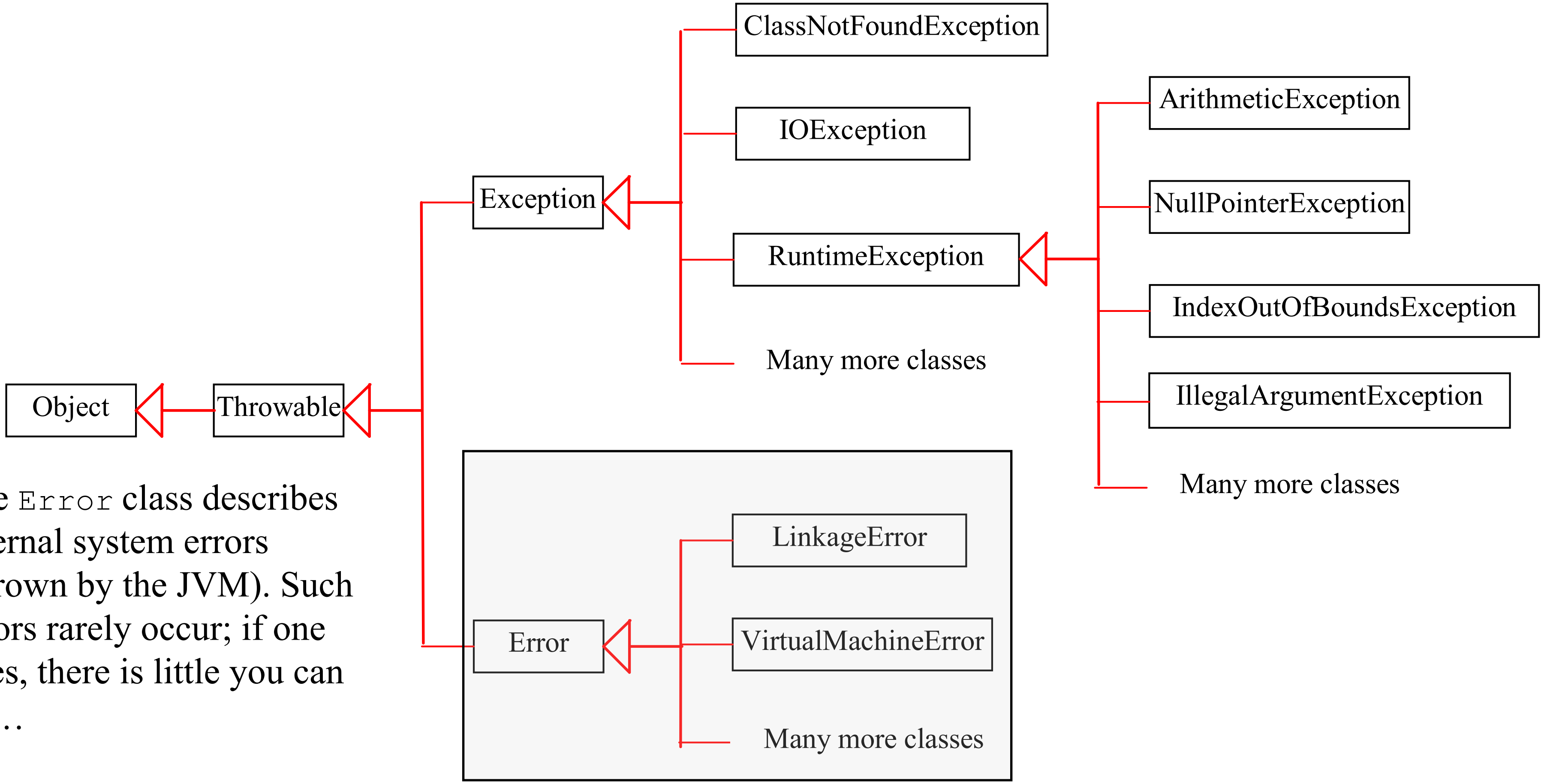
```
public class Divide2 {

   public static int divide(int n1, int n2) throws ArithmeticException {
        if (n2 == 0)
           throw new ArithmeticException("Divisor cannot be zero");

        return n1 / n2;
   }

   public static void main(String[] args) {
        int x = 1;
        int y = 0;

        try {
           int result = divide(x, y);
           System.out.println("result: " + result);
        }
        catch (ArithmeticException ex) {
           System.out.println("Exception: integer division by zero");
        }

        System.out.println("Execution continues ...");
   }
}
```

If an `ArithmeticException` occurs

Exception Types



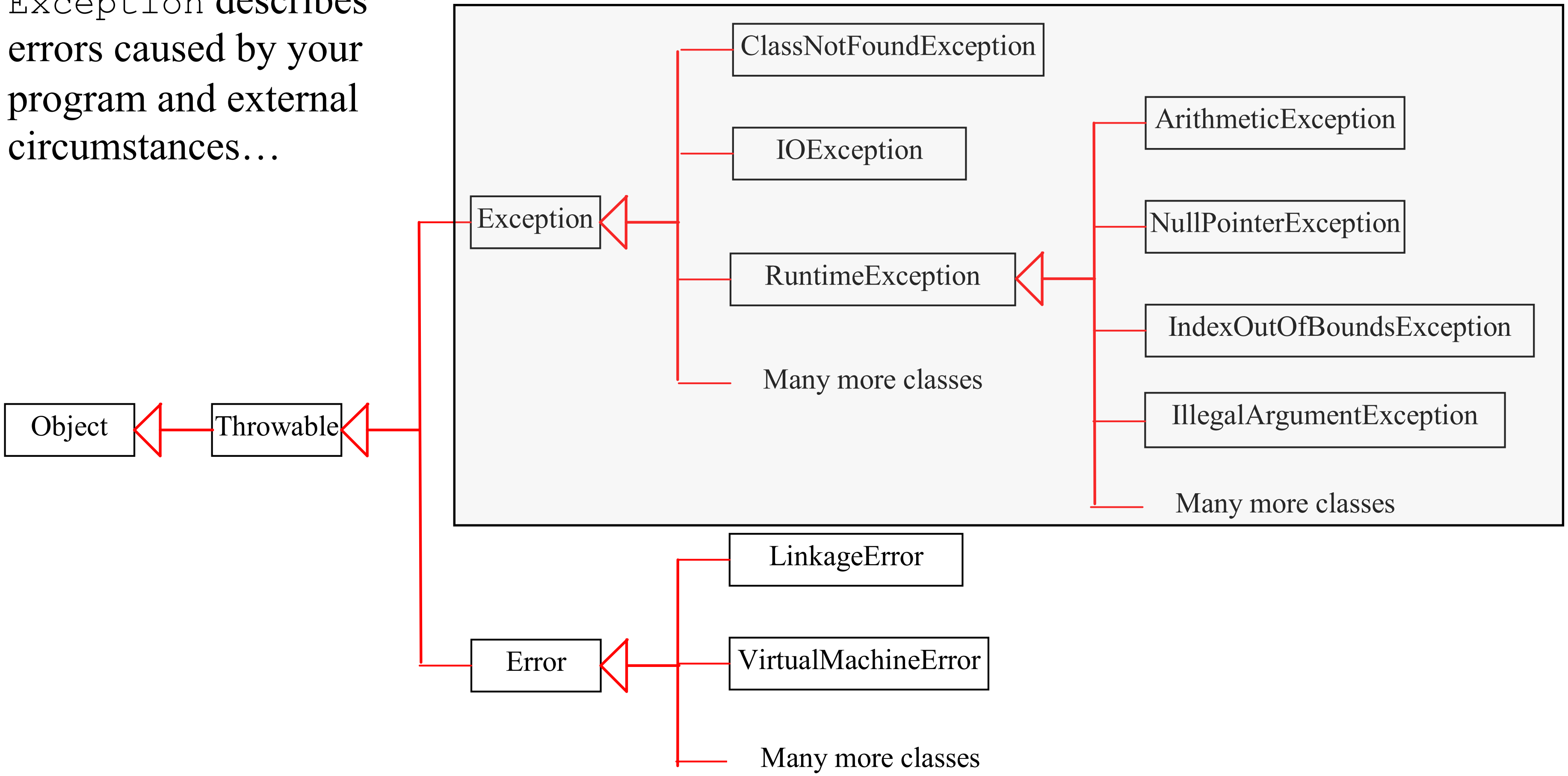
System Errors



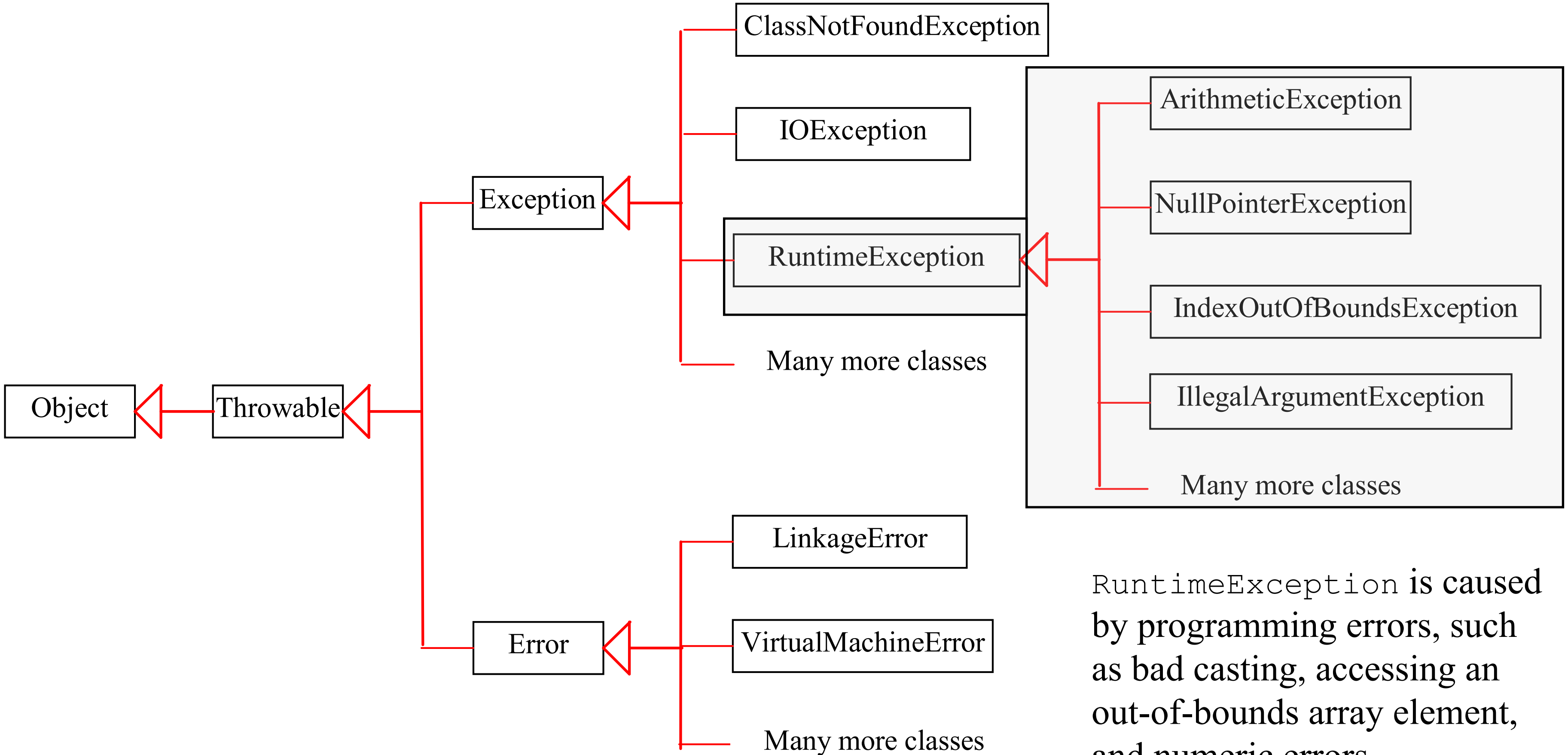
The `Error` class describes internal system errors (thrown by the JVM). Such errors rarely occur; if one does, there is little you can do…

Exceptions

`Exception` describes errors caused by your program and external circumstances…



Runtime Exceptions



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

Checked vs. Unchecked Exceptions

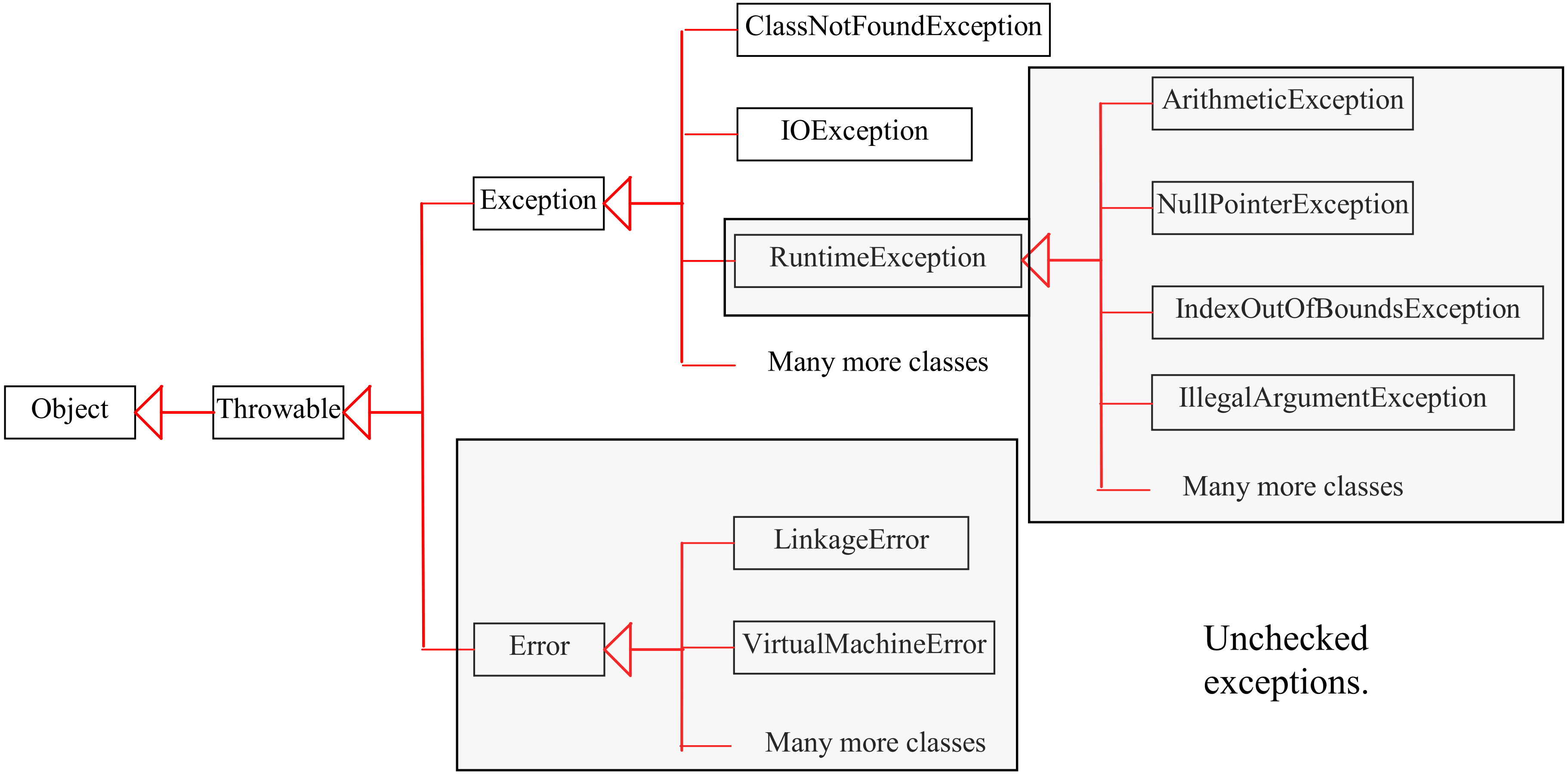
`RuntimeException`, `Error` and their subclasses are known as *unchecked exceptions*:

- In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. These errors should be corrected in the program.
- For example, an `IndexOutOfBoundsException` is thrown if you access an element in an array outside the bounds of the array.
- To avoid cumbersome overuse of try-catch blocks, Java does not mandate you to catch unchecked exceptions.

All other exceptions are known as *checked exceptions*:

- The compiler forces the programmer to check and deal with these exceptions.

Unchecked Exceptions



Unchecked exceptions.

Exception Handling Model

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

Declaring Exceptions

All (checked) exceptions that might be thrown by a method must be explicitly declared in the method header.

To *declare* an exception (or exceptions) in a method, use the `throws` keyword in the method header:

```
public void myMethod() throws Ex1, Ex2, ..., ExN {
   // some statements
}
```

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*.

```
public static int divide(int n1, int n2) throws ArithmeticException {
   if (n2 == 0)
        throw new ArithmeticException("Divisor cannot be zero");

   return n1 / n2;
}
```

Catching Exceptions

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

```
try {
   statements; // may throw Exception1, Exception2 … ExceptionN
   ...
}
catch (Exception1 ex1) {
   handler for exception1;
}
catch (Exception2 ex2) {
   handler for exception2;
}
...
catch (ExceptionN exN) {
   handler for exceptionN;
}
```

Catching Exceptions

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

```
try {
   statements; // may throw Exception1, Exception2 … ExceptionN
   ...
}
catch (Exception1 ex1) {
   handler for exception1;
}
catch (Exception2 ex2) {
   handler for exception2;
}
...
catch (ExceptionN exN) {
   handler for exceptionN;
}
```

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

The `finally` Clause

Occasionally, you may want some code to be executed regardless of whether or not an exception occurs.

The `finally` clause is always executed whether an exception occurs or not.

Example:

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

The `finally` Clause – Example #1

animation

Trace a Program Execution

```
try {
  statements;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

animation

Trace a Program Execution

Suppose no exceptions are thrown in the statements

```
try {
  statements;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

animation

Trace a Program Execution

```
try {
  statements;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed

animation

Trace a Program Execution

```
try {
  statements;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The next statement in the method is executed

The `finally` Clause – Example #2

animation

Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

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

Getting Information from Exceptions

The `java.lang.Throwable` class is the root class for all exception objects.

It contains several methods to get information about an exception.

These methods are useful for debugging.

chapter12_examples_1

Divide3

Example: Declaring, Throwing, and Catching Exceptions

This example demonstrates declaring, throwing, and catching exceptions by modifying the `setRadius` method in the `Circle` class. Constructors invoke the `setRadius` method. The new `setRadius` method throws an exception if the radius is negative.

(Alternatively, in the constructors and `setRadius` method, can simply set the radius to e.g. 0 if the specified radius is negative…)

chapter12_examples_1

Circle

TestCircle1

TestCircle2

Benefits of Exception Handling

Exceptions are thrown from a method:

- In general, you should not let the method in which the exception occurs (referred to as the called method) handle the exception.
- Why? Often the called method does not know what to do in case of error. This is typically the case for the library methods. Library methods can detect errors, but only the caller knows what needs to be done when an error occurs.
- The caller of the method should decide how to handle the exception (e.g. whether to terminate the program or continue its execution).

Exception handling enables a method to throw an exception to its caller, thereby enabling the caller to handle the exception.

Benefits of Exception Handling

For example, consider reading an integer from the standard input:

- The `nextInt` method in the `Scanner` class can be used to read the integer.
- This method will throw an exception (`InputMismatchException`) if the input is invalid.
- If the input is invalid, the calling method should handle the exception – it should not be handled by the `nextInt` method because the appropriate action to take depends on the particular application.

The key benefit of exception handling is separating the detection of an error (done in a called method) from the handling of an error (done in the calling method).

Examples – reading an integer using the Scanner class using:

(a) exception handling

(b) the `hasNextInt` method defined in the `Scanner` class

chapter12_examples_1

ReadNumber1

ReadNumber2

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 (e.g. checking for integer division by zero).

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 (e.g. checking for integer division by zero).

For example, the following code…

```
Circle c = new Circle();
...
try {
   System.out.println(c.toString());
} catch (NullPointerException ex) {
   System.out.println("c is null");
}
```

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 (e.g. checking for integer division by zero).

For example, the following code…

```
Circle c = new Circle();
...
try {
   System.out.println(c.toString());
} catch (NullPointerException ex) {
   System.out.println("c is null");
}
```

…is better written as…

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

Text I/O

The `java.io.File` class contains methods for obtaining the properties of a file/directory and for renaming and deleting a file/directory.

Note that creating a `File` instance does not create a file on a machine:

- You can create a `File` instance for any file name regardless of whether it exists or not.
- You can invoke the `exists()` method on a `File` instance to check whether the file exists.

A `File` object does not contain the methods for reading/writing data from/to a file:

- In order to perform file I/O, you need to create objects using appropriate Java I/O classes.
- The `Scanner` and `PrintWriter/FileWriter` classes can be used to read/write from/to a text file.

Text I/O – Examples

Example #1 – how to create a `File` object and use the methods in the `File` class to obtain the properties of a file/directory.

TestFile

Example #2 – how to write data to a text file using the `PrintWriter` and `FileWriter` classes.

WriteData

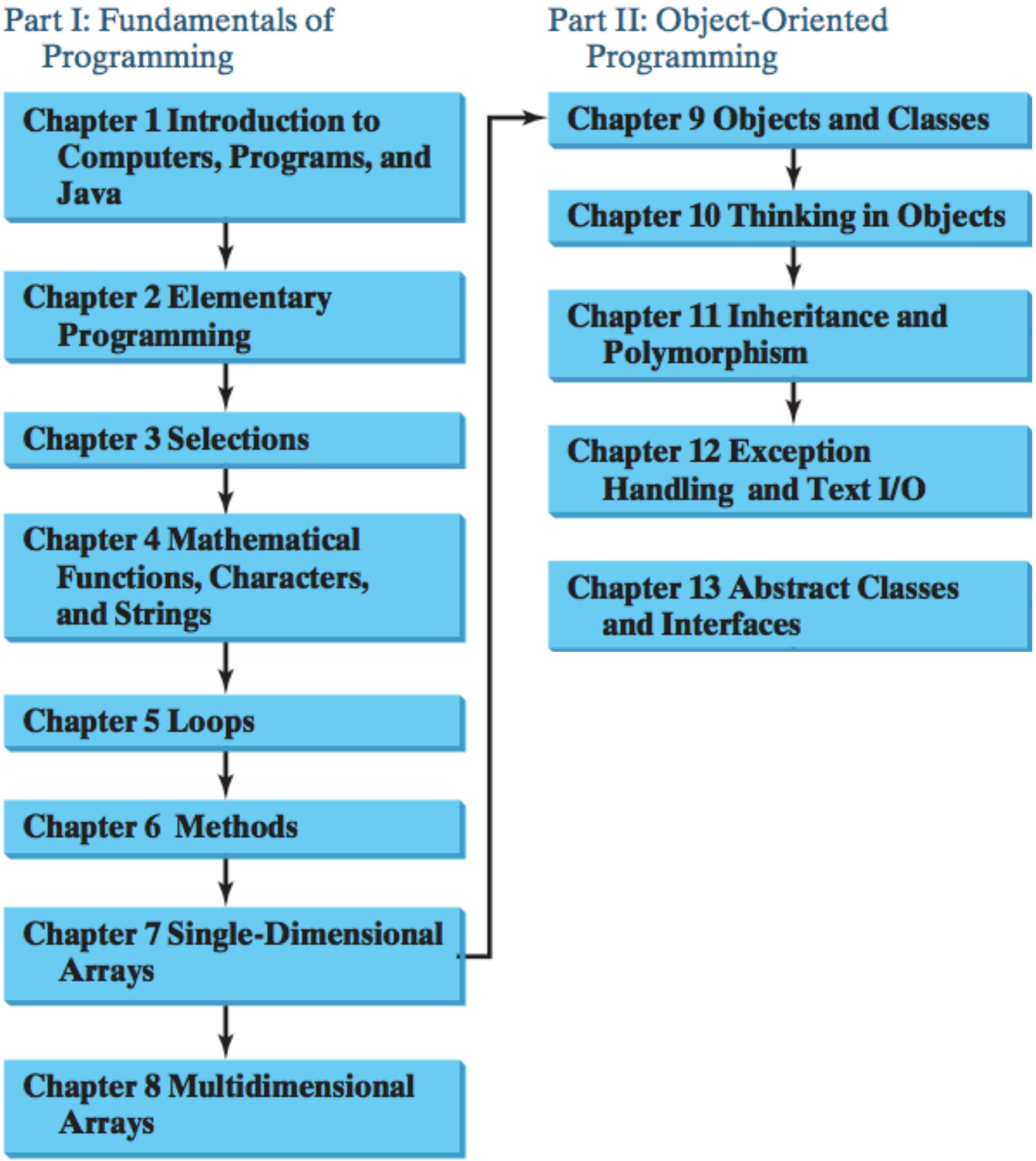
Example #3 – how to read data from a text file using the `Scanner` class.

ReadData

chapter12_examples_2

This Lecture…

In this lecture, we covered exception handling and text I/O...



Part I: Fundamentals of Programming

Chapter 1 Introduction to Computers, Programs, and Java

Chapter 2 Elementary Programming

Chapter 3 Selections

Chapter 4 Mathematical Functions, Characters, and Strings

Chapter 5 Loops

Chapter 6 Methods

Chapter 7 Single-Dimensional Arrays

Chapter 8 Multidimensional Arrays

Part II: Object-Oriented Programming

Chapter 9 Objects and Classes

Chapter 10 Thinking in Objects

Chapter 11 Inheritance and Polymorphism

Chapter 12 Exception Handling and Text I/O

Chapter 13 Abstract Classes and Interfaces