Chapter 6
EXCEPTION HANDLING AND TEXT I/O

1 Introduction

- An *exception* is an object that represents an error or a condition that prevents execution from proceeding normally.
- How can you handle the exception so that the program can continue to run or else terminate gracefully?

2 Exception-Handling Overview

- Exceptions are thrown from a method. The caller of the method can catch and handle the exception.

```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));
  }
}
```

2 Exception-Handling Overview

```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 ");
  }
}
```

```java
import java.util.Scanner;

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

    return number1 / number2;
  }

  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);
  }
}
```

```java
import java.util.Scanner;

public class QuotientWithException {
  public static int quotient(int number1, int number2) {
    if (number2 == 0)
      throw new ArithmeticException("Divisor cannot be zero");

    return number1 / number2;
  }

  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 ...");
  }
}
```

If an Arithmetic Exception occurs

2 Exception-Handling Overview

In summary, a template for a try-throw-catch block may look like this:

```
try {
        Code to run;
        A statement or a method that may throw an exception;
        More code to run;
}catch (type ex) {
        Code to process the exception;
 }
```

2 Exception-Handling Overview

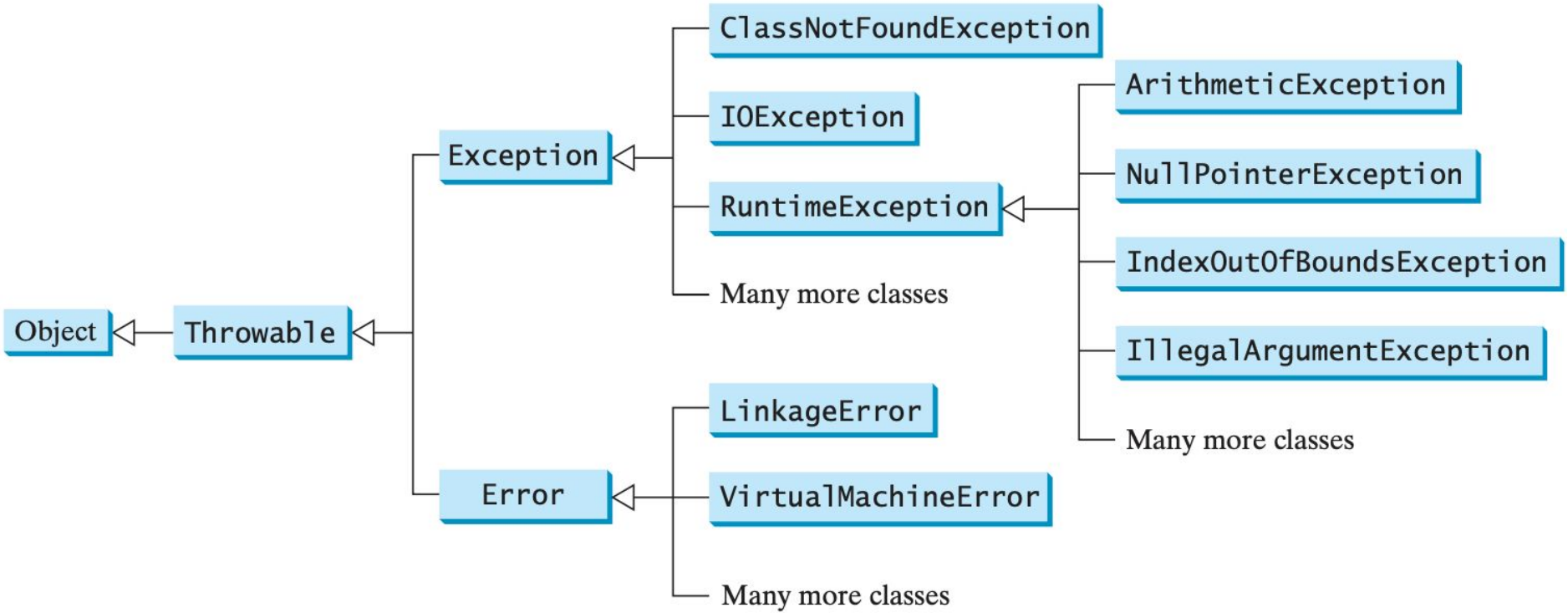
Key benefit: separating the detection of an error (done in a called method) from the handling of an error (done in the calling method).

- nextInt() throw InputMismatchException
- the example handles the exception when reading an input.

```java
import java.util.*;

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);
  }
}
```

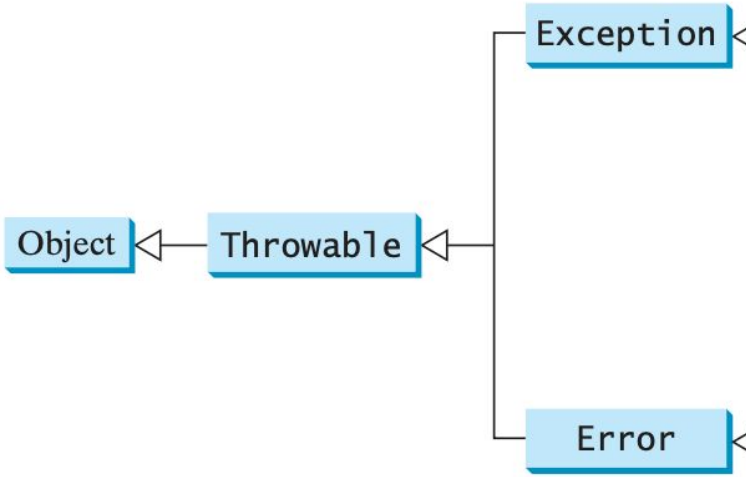
If an InputMismatchException occurs

3 Exception Types

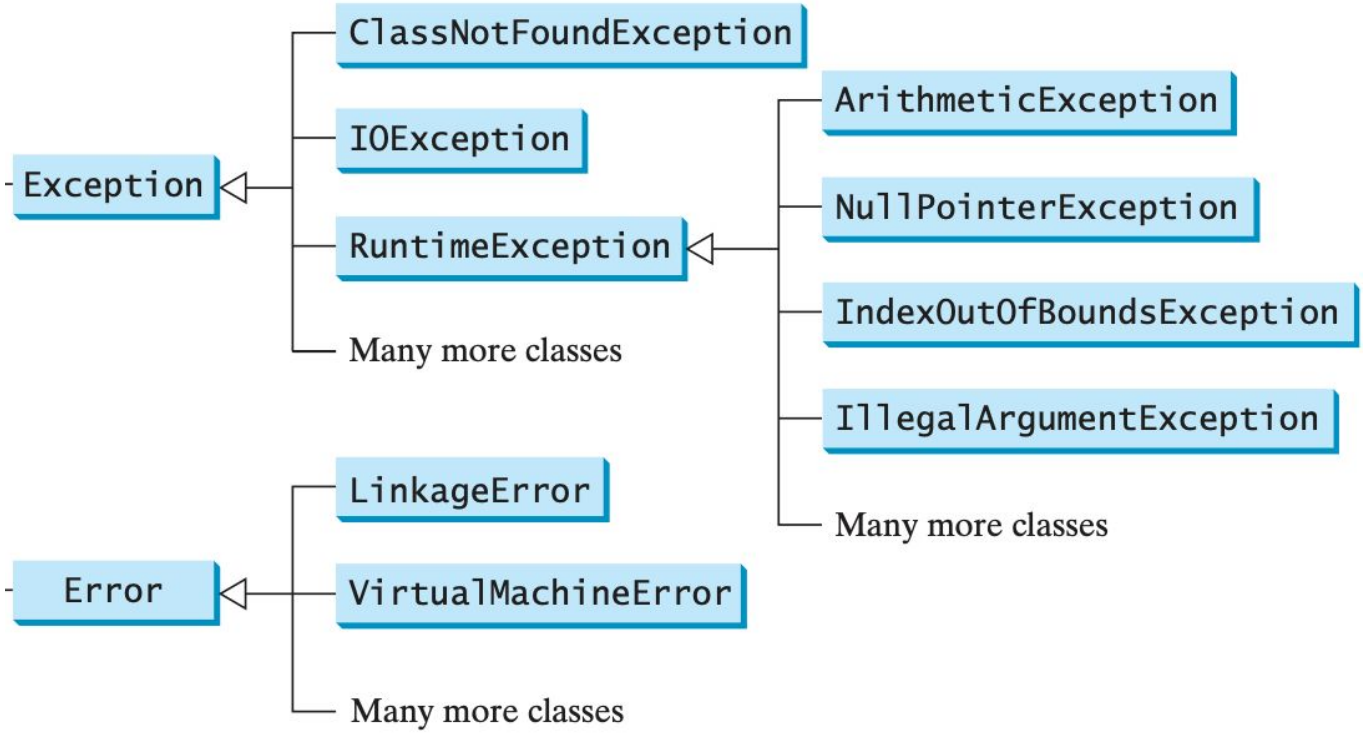


3 Exception Types

- **Exception**: errors caused by your program and by external circumstances.
- **Error (**or system errors**)**: internal errors and resource exhaustion situations inside the Java runtime system => should not throw an object of this type



3 Exception Types

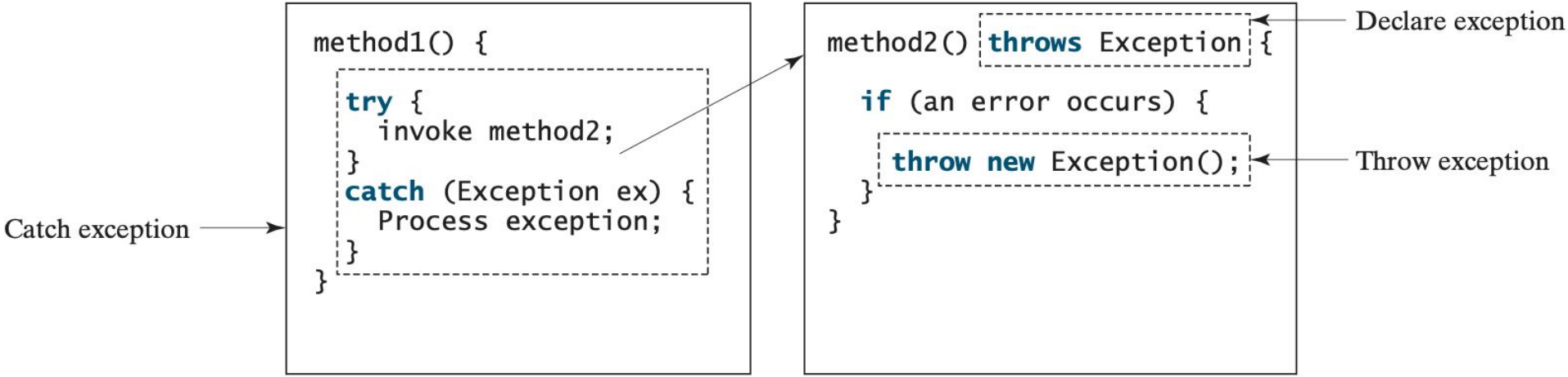


3 Exception Types

- **Unchecked exception**: Error, RuntimeException ,and their subclasses
  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.
- **Checked exception**: others
  The compiler requires that you provide exception handlers for all checked exceptions.

4 More on Exception Handling

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



4 More on Exception Handling

- Declaring Exceptions
  public void myMethod() **throws** Exception1, Exception2, ..., ExceptionN

- Throwing Exceptions
  throw new IllegalArgumentException("Wrong Argument");

- Catching Exceptions: catch multiple exception types in the same catch clause when they are not subclasses of one another

```
try {
    //code that might throw exceptions
} catch (FileNotFoundException | UnknownHostException e) {
    // emergency action for missing files and unknown hosts
} catch (IOException e) {
    // emergency action for all other I/O problems
}
```

4 More on Exception Handling

Getting Information from Exceptions

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.

5 The finally Clause

- The finally clause is always executed regardless whether an exception occurred.
- The finally block executes even if there is a return statement prior to reaching the finally block.

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

```
try {
        statements;
}finally {
        finalStatements;
}
```

5 The finally Clause

```
InputStream in = new FileInputStream(. . .);
try{
        //1
        code that might throw exceptions
        //2
}catch (IOException e) {
        //3
        show error message
        //4
}finally {
        //5
        in.close();
}
// 6
```

- Throws no exceptions
- Throws an exception that is caught in a catch clause

6 When to Use Exceptions

A method should throw an exception if the error needs to be handled by its caller.
Do not to abuse exception handling as a way to deal with a simple logic test.

```
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");
```

7 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 exits;
    throw ex;
}
```

8 Defining Custom Exception Classes

- You can define a custom exception class by extending the java.lang.Exception class.
- Use built-in exception classes whenever possible instead of defining your own exception classes.

  TestCircleWithCustomException.java, page 471 textbook

8 Defining Custom Exception Classes

```
public class InvalidRadiusException extends Exception {
   private double radius;
    /** Construct an exception */
   public InvalidRadiusException(double radius) {
     super("Invalid radius " + radius);
     this.radius = radius;
   }
    /** Return the radius */
   public double getRadius() {
     return radius;
   }
 }
```

9 Using Assertions

7.4.1 The Assertion Concept

```
double y = Math.sqrt(x);  // x > 0

if (x < 0) throw new IllegalArgumentException("x < 0");

// this code stays in the program, even after testing is complete
// run quite a bit slower if lots of checks like throw statements.
```

9 The Assertion Concept

The assertion mechanism allows to:

- check during testing
- automatically removed in the production code

Two forms: evaluate the condition and throw an AssertionError (and expression) if it is false

```
assert condition;
```

and

```
assert condition: expression;
```

E.g,

```
assert x >= 0;
assert x >= 0 : x;
```

9 Assertion Enabling and Disabling

- By default, assertions are disabled, the class loader strips out the assertion code so that it won't slow execution.
- Enable by:

```
java -enableassertions MyApp
java -ea MyApp
```

10 The File Class

- The File class contains the methods for obtaining file and directory properties and for renaming and deleting files and directories.

java.io.File

```
+File(pathname: String)

+File(parent: String, child: String)

+File(parent: File, child: String)

+exists(): boolean
+canRead(): boolean
+canWrite(): boolean
+isDirectory(): boolean
+isFile(): boolean
+isAbsolute(): boolean
+isHidden(): boolean
+getAbsolutePath(): String

+getCanonicalPath(): String

+getName(): String

+getPath(): String

+getParent(): String

+lastModified(): long
+length(): long
+listFile(): File[]
+delete(): boolean

+renameTo(dest: File): boolean

+mkdir(): boolean

+mkdirs(): boolean
```

10 File Input and Output

Use the Scanner class for reading text data from a file and the PrintWriter class for writing text data to a file.

java.io.PrintWriter	
`+PrintWriter(file: File)`	Creates a `PrintWriter` object for the specified file object.
`+PrintWriter(filename: String)`	Creates a `PrintWriter` object for the specified file-name string.
`+print(s: String): void`	Writes a string to the file.
`+print(c: char): void`	Writes a character to the file.
`+print(cArray: char[]): void`	Writes an array of characters to the file.
`+print(i: int): void`	Writes an `int` value to the file.
`+print(l: long): void`	Writes a `long` value to the file.
`+print(f: float): void`	Writes a `float` value to the file.
`+print(d: double): void`	Writes a `double` value to the file.
`+print(b: boolean): void`	Writes a `boolean` value to the file.
Also contains the overloaded `println` methods.	A `println` method acts like a `print` method; additionally, it prints a line separator. The line-separator string is defined by the system. It is `\r\n` on Windows and `\n` on Unix.
Also contains the overloaded `printf` methods.	The `printf` method was introduced in §4.6, "Formatting Console Output."

```java
public class WriteData {
   public static void main(String[] args) throws java.io.IOException {
     java.io.File file = new java.io.File("scores.txt");
     if (file.exists()) {
       System.out.println("File already exists");
       System.exit(0);
     }
      // Create a file
     java.io.PrintWriter output = new java.io.PrintWriter(file);
      // Write formatted output to the file
     output.print("John T Smith "); output.println(90);
     output.print("Eric K Jones "); output.println(85);
      // Close the file
     output.close();
   }
 }
```

10 File Input and Output

Closing Resources Automatically Using try-with-resources

```
public class WriteDataWithAutoClose {
  public static void main(String[] args) throws Exception {
    java.io.File file = new java.io.File("scores.txt");
    if (file.exists()) {
      System.out.println("File already exists");
      System.exit(0);
    }

    try (
      // Create a file
      java.io.PrintWriter output = new java.io.PrintWriter(file);
    ) {
      // Write formatted output to the file
      output.print("John T Smith ");
      output.println(90);
      output.print("Eric K Jones ");
      output.println(85);
    }
  }
}
```

10 File Input and Output

Reading Data Using Scanner

```
import java.util.Scanner;

public class ReadData {
  public static void main(String[] args) throws Exception {
    // Create a File instance
    java.io.File file = new java.io.File("scores.txt");

    // Create a Scanner for the file
    Scanner input = new Scanner(file);

    // Read data from a file
    while (input.hasNext()) {
      String firstName = input.next();
      String mi = input.next();
      String lastName = input.next();
      int score = input.nextInt();
      System.out.println(
        firstName + " " + mi + " " + lastName + " " + score);
    }

    // Close the file
    input.close();
  }
}
```