Review Error Handling Binary I/O

Dr. Abdallah Karakra |Comp 2311 | Masri504

5/24/2023

BIRZEIT UNIVERSITY

Index

Chapter 12

- Revision.
- Section 12.8.
- Section 12.11.

Chapter 17

- Section 17.2
- Section 17.3
- Section 17.4
- Section 17.5
- Section 17.7

CHAPTER 12

Review Error Handling

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

Motivations

- When a Java program runs into an unexpected runtime error, the program terminates *abnormally*.

- How can you handle these events so that the program can (under your control) *continue to run* or *terminate gracefully*?

- The topics in this chapter will allow you to create IDEA stronger, more resilient programs that react well to abnormal execution-time situations.

Exception-Handling Overview

Example – Scenario1

The following three code samples directly perform a division operation (*CAUTION: Dividing by zero is an undefined operation!*)

1. Naive code – no protection
2. Fix it using an *if statement*
3. Fix-it with *Exception* handler.

Scenario2. *What if the runtime error occurs in a called method?*

Exception-Handling Overview 1 of 3

```
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));
  }
}
```

1. Naive code
No protection

CONSOLE

```
Enter two integers: 2 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Quotient.main(Quotient.java:12)
```

RED text messages are runtime ERRORS caught by the JVM

Exception-Handling Overview 2 of 3

```
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 ");
  }
}
```

2. Protect code with if-stm

CONSOLE

```
Enter two integers: 2 0
Divisor cannot be zero
```

Exception-Handling Overview 3 of 3

```
import java.util.Scanner;
Import java.lang.ArithmeticException;
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,number2;
    try {
         int number1 = input.nextInt();
         int number2 = input.nextInt();

         int result = 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 ...");
  }
}
```

3. Protect code with Exception

CONSOLE

```
Enter two integers: 2 0
Exception: an integer cannot be divided by zero
Execution continues ...
```

Introduction

- An exception is an object that represents an error or a condition that prevents execution from proceeding normally.

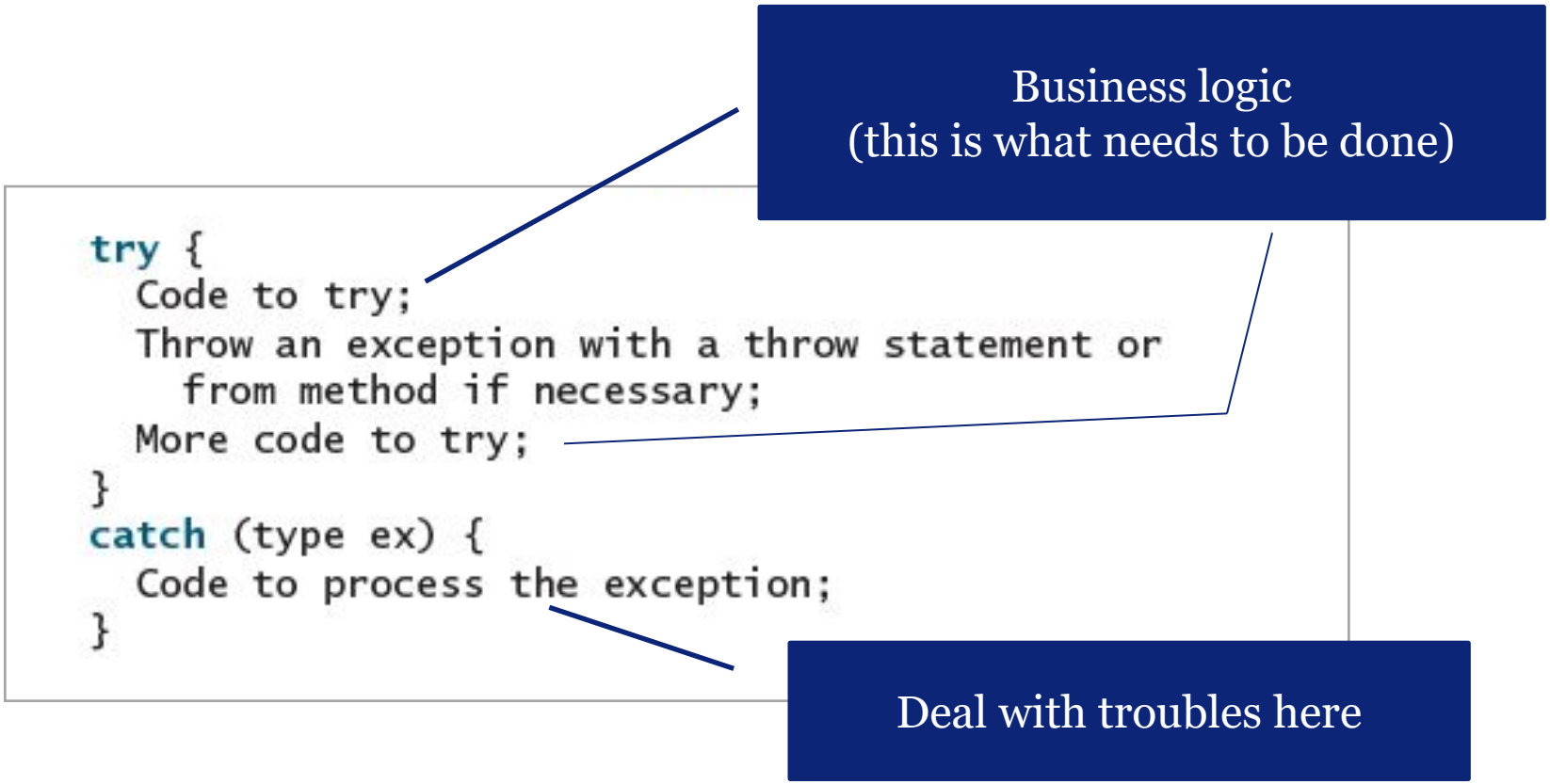
- Exception handling enables a program to deal with exceptional situations and continue its normal execution.

Exception Handling

Keywords:

Exception-Handling Advantage

Exception handling **separates error-handling code** from **normal programming tasks**, consequently making programs *easier to read* and to *modify*.



Exception-Handling Overview

Some (of the many) pre-defined Java **Exceptions**

```
ArithmeticException
ClassNotFoundException
IllegalAccessException
IOException
EOFException
FileNotFoundException
InputMismatchException
MalformedURLException
ProtocolException
SocketException
UnknownHostException
UnknownServiceException
. . .
```

Example: Handling InputMismatchException

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

```
public class InputMismatchExceptionDemo {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    boolean continueInput = true;

    while (continueInput) {

      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
      }

    }
  }
}
```

```
CONSOLE
Enter an integer: 55.55
Try again. (Incorrect input: an integer is required)
Enter an integer: 66
The number entered is 66
```

Example: Handling InputMismatchException

Import the required packages

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

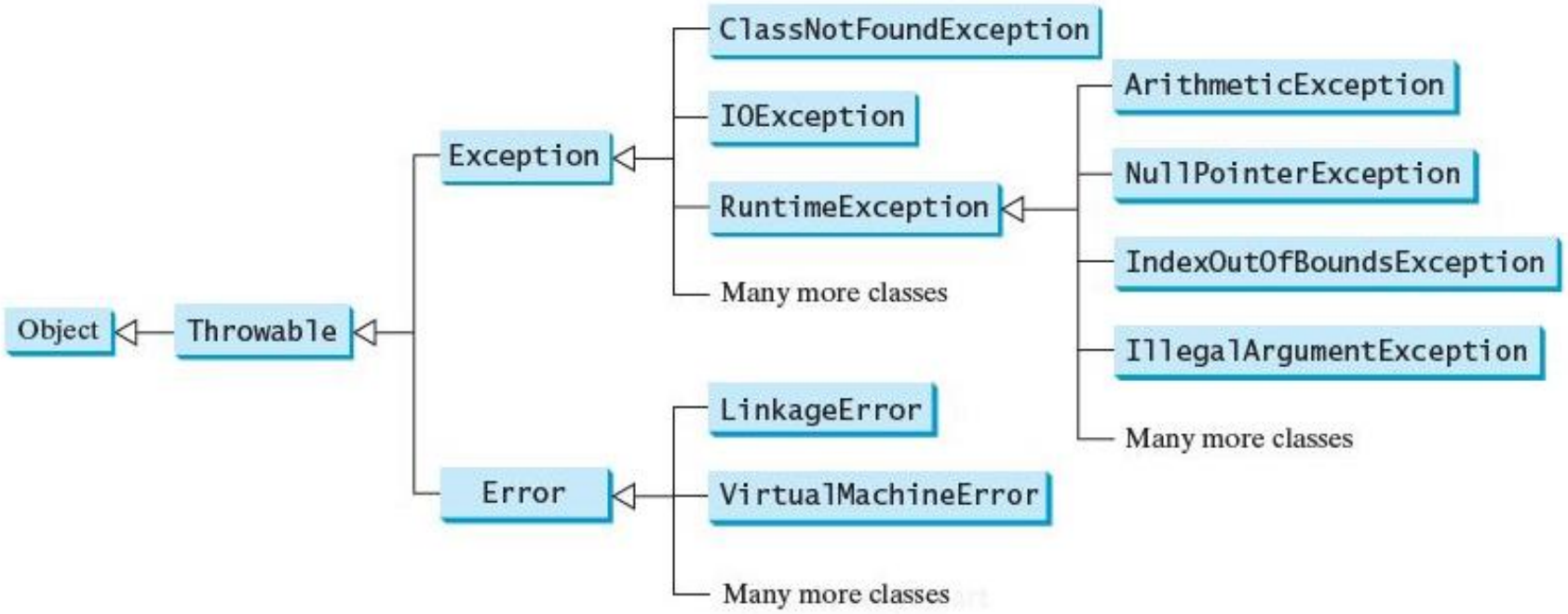
```
public class InputMismatchExceptionDemo {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    boolean continueInput = true;
    while (continueInput) {

      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
      }

    }
  }
}
```

Why should I? Find out what the answer is.

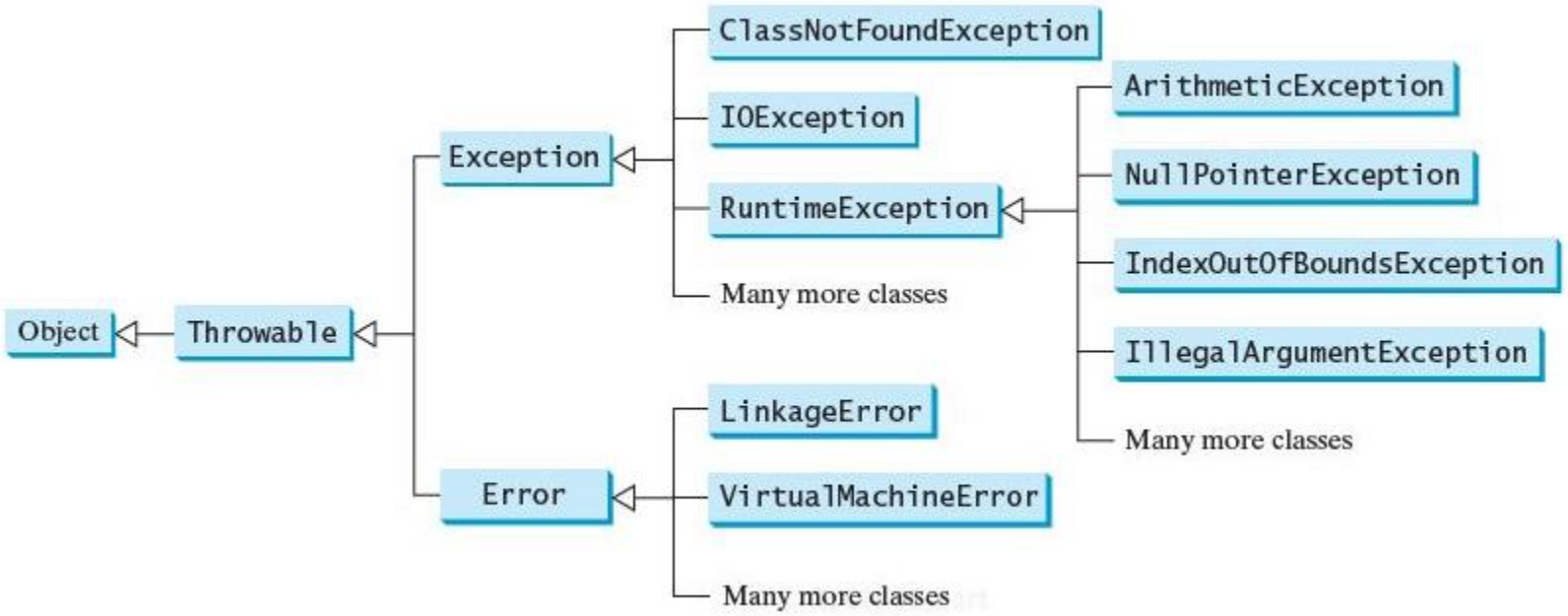
```
CONSOLE
Enter an integer: 55.55
Try again. (Incorrect input: an integer is required)
Enter an integer: 66
The number entered is 66
```

Exception Types

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



Exception Types



System errors are thrown by JVM.

There is little you can do beyond notifying the user and trying to terminate the program gracefully.

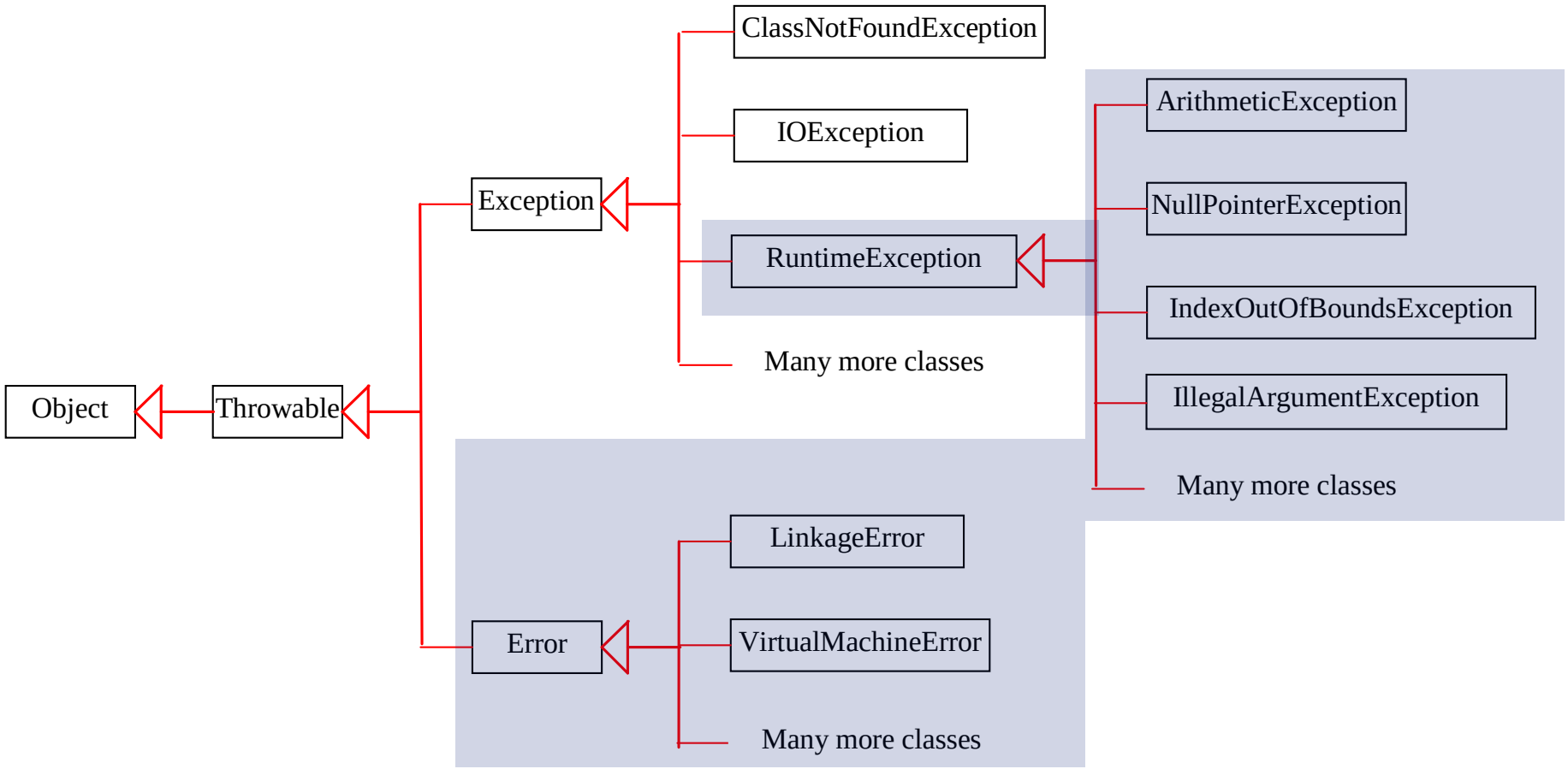
Exception describes errors caused by your program and external circumstances.

These errors can be caught and handled by your program.

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 Exceptions

- In most cases, unchecked exceptions reflect programming-logic errors that are not recoverable (poor logic, bad programming,...) For example
  - `NullPointerException,`
  - `IndexOutOfBoundsException`
- Unchecked exceptions can occur anywhere in the program.
- Java does not mandate you to write code to catch unchecked exceptions *(bad code happens!)*.

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
```

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*. Here is an example:

```
throw new  MyNewException(optionalMsg);
```

```
MyNewException ex = new MyNewException();

throw ex(optionalMsg);
```

Throwing 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");

 }
```

Using a pre-defined exception

Catching Exceptions

```
try {
  statements;  // Statements that may throw exceptions
}
catch (Exception1 ex1) {
  handler for exception1;
}
catch (Exception2 ex2) {
  handler for exception2;
}
...
catch (ExceptionN exn) {
  handler for exceptionN;
}
```

Catching multiple exceptions (*one-at-the-time*)

Catching Exceptions

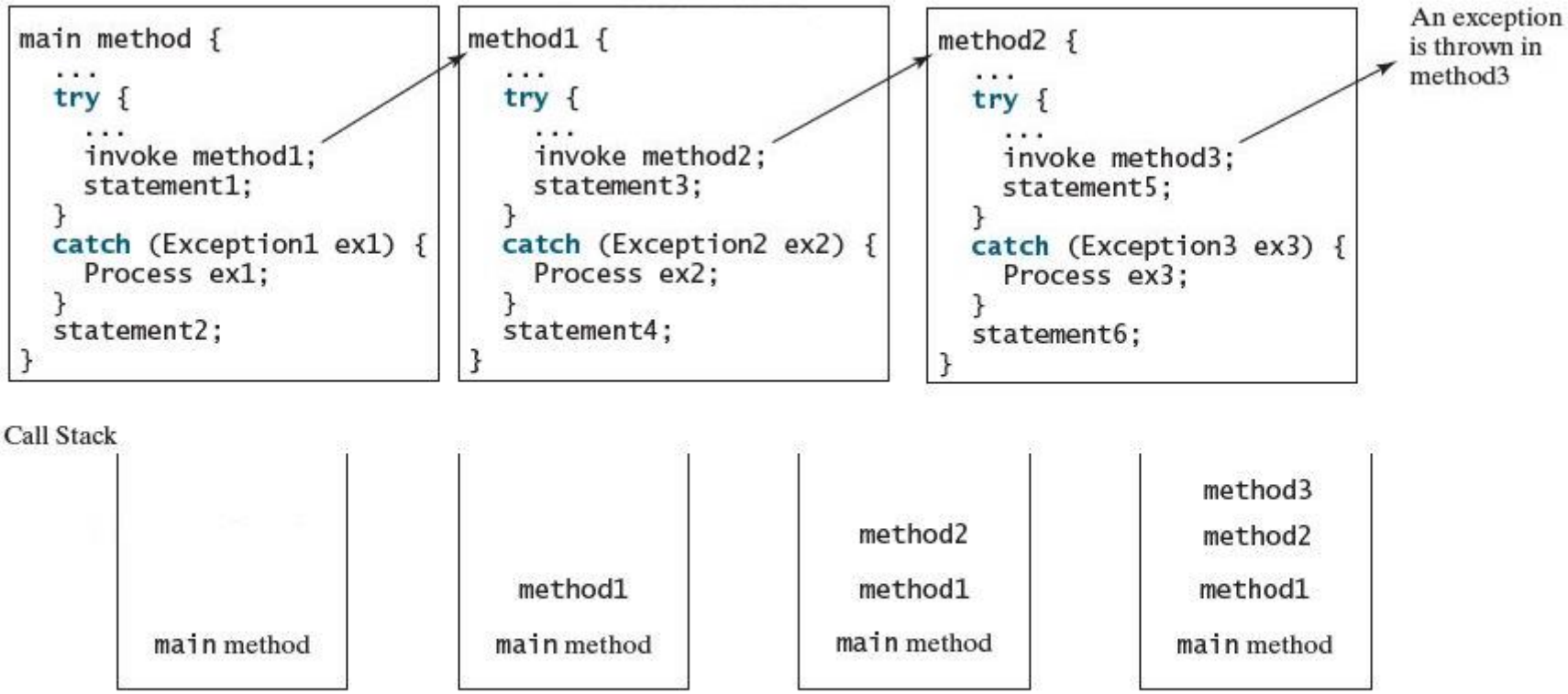


FIGURE 13.3 If an exception is not caught in the current method, it is passed to its caller. The process is repeated until the exception is caught or passed to the `main` method.

Example: Declaring, Throwing, and Catching Exceptions

Objective:

- This example demonstrates declaring, throwing, and catching exceptions by modifying the `setRadius` method in the `Circle` class.

- The new `setRadius` method throws an exception if radius is negative.

Example: Declaring, Throwing, and Catching Exceptions 1 of 2

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

  /** Construct a circle with a specified radius */
  public CircleWithException(double newRadius) {
    setRadius(newRadius);
  }
  /** Construct a circle with radius 1 (Default)*/
  public CircleWithException() {
    this(1.0);
  }

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

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

Example: Declaring, Throwing, and Catching Exceptions 2 of 2

```
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());
  }
}
```

Example: Declaring, Throwing, and Catching Exceptions 2 of 2

```
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);

    atch (IllegalArgumentException ex) {
      System.out.println(ex);
    }

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

write an implementation for this method

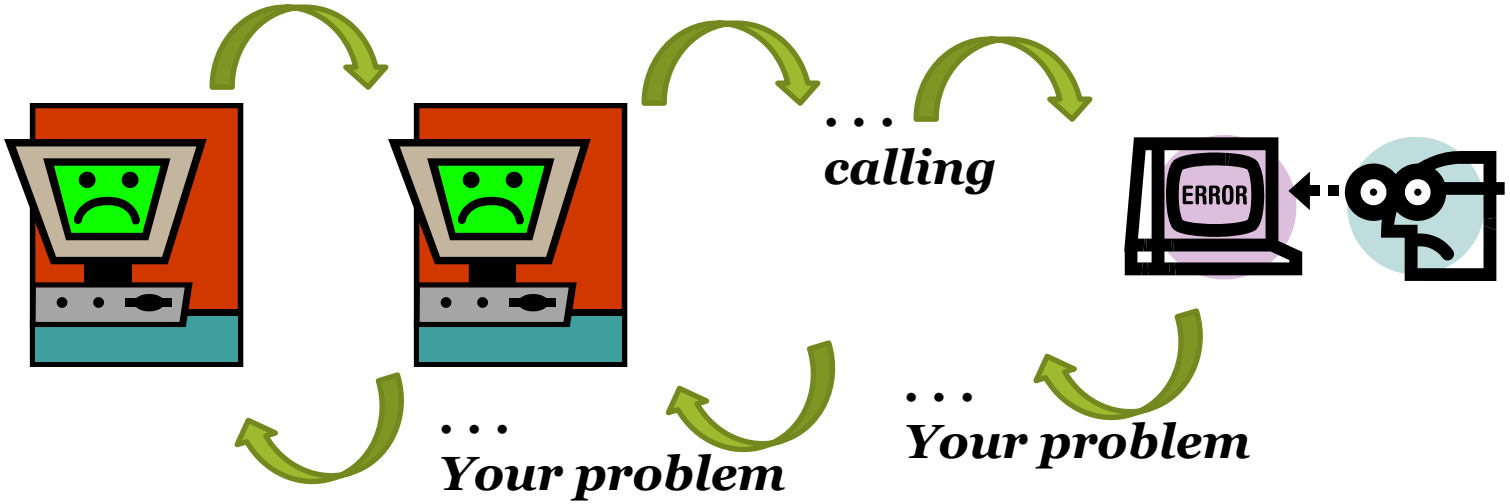
What is the expected number of returned objects?

Example

```
public static void writeToFile() throws IOException {

    BufferedWriter bw = new BufferedWriter(new FileWriter("myFile.txt"));
    bw.write("Test"); bw.close();

}

 public static void main(String[] args)
 {
    try {
            writeToFile();
    } catch (IOException ioe) {
              ……….
    }

}
```

Exception Management Advantages

- The Exception-mechanism enables a called method to demand the *strongest attention* from its caller (by performing a throw-statement).
- *Without this capability, the called method must handle the problem or terminate the program (consider the division problem)*



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( SomeException ex) {
  perform some operations here;
  throw ex;
}
```

Example: Rethrowing Exceptions

```
public class Test {
  public int testMethod(int n1, int n2) {
    try {
      return n1 / n2;
    }
    catch (ArithmeticException e) {
      throw e;
    }
  }

  public static void main(String[] args) {
    Test obj = new Test();
    try {
      System.out.println(obj.testMethod(30, 0));
    }
    catch (Exception e) {
      System.out.println("process the exception here");
    }
  }
}
```

CONSOLE
```
process the exception here
```

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.

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

Cautions When Using Exceptions

- Exception handling usually consumes 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.

However (in general) the benefits out-weight the risks !

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 it. A simple `if-statement` should be sufficient.

Defining Custom Exception Classes

- Define custom exception classes if the predefined classes are not sufficient.
- *Define custom exception classes by extending Exception or a subclass of Exception*.

Example: Defining Custom Exceptions

Defining a custom exception for rejecting a negative radius value (**Note**: predefined `IllegalArgumentException` could had been used instead)

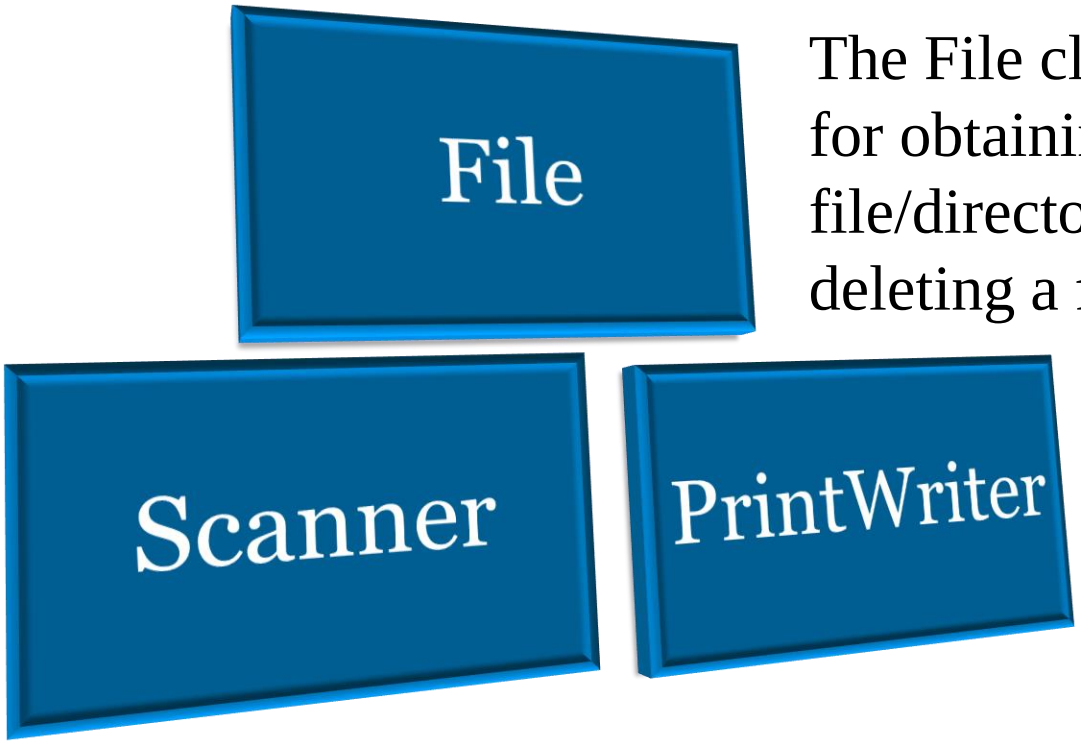
```
public class MyInvalidRadiusException extends Exception {

    private String myMsg = "";

    public MyInvalidRadiusException(String userMsg) {
        // user-defined message
        myMsg = userMsg;
    }

    public MyInvalidRadiusException() {
        // default message
        myMsg = "Invalid RADIUS. It must be a positive value";
    }

    @Override
    public String getMessage() {
        return myMsg;
    }
}
```

Example: Defining Custom Exceptions

This is a fragment of the Circle2 class throwing the custom exception

```
public class Circle2 {
   private double radius;

   public Circle2() throws MyInvalidRadiusException{
      setRadius(0);
   }

   public Circle2(double radius) throws MyInvalidRadiusException{
      setRadius(radius);
   }

   /** Set a new radius - it must be a positive number
    * @throws Exception */
   public void setRadius(double radius) throws MyInvalidRadiusException{
      if ( radius >= 0)
         this.radius = radius;
      else
         throw new MyInvalidRadiusException("Radius must be positive "
                                            + radius);
   }

   . . .

}
```

Read/Write from/to file



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

To read/write from/to afile

Writing Data Using PrintWriter

java.io.PrintWriter	
+PrintWriter(filename: String)	Creates a PrintWriter for the specified file.
+print(s: String): void	Writes a string.
+print(c: char): void	Writes a character.
+print(cArray: char[]): void	Writes an array of character.
+print(i: int): void	Writes an int value.
+print(l: long): void	Writes a long value.
+print(f: float): void	Writes a float value.
+print(d: double): void	Writes a double value.
+print(b: boolean): void	Writes a boolean value.
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 and Strings."

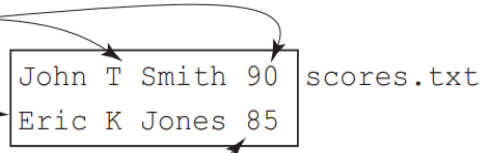
print, println, and printf

File Input and Output

12.11.1 Writing Data Using PrintWriter

WriteData.java

```
1  public class WriteData {
2    public static void main(String[] args) throws java.io.IOException {
3      java.io.File file = new java.io.File("scores.txt");
4      if (file.exists()) {
5        System.out.println("File already exists");
6        System.exit(1);
7      }
8
9      // Create a file
10     java.io.PrintWriter output = new java.io.PrintWriter(file);
11
12     // Write formatted output to the file
13     output.print("John T Smith ");
14     output.println(90);
15     output.print("Eric K Jones ");
16     output.println(85);
17
18     // Close the file
19     output.close();
20   }
21 }
```



Reading Data Using Scanner

java.util.Scanner	
+Scanner(source: File)	Creates a Scanner object to read data from the specified file.
+Scanner(source: String)	Creates a Scanner object to read data from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has another token in its input.
+next(): String	Returns next token as a string.
+nextByte(): byte	Returns next token as a byte.
+nextShort(): short	Returns next token as a short.
+nextInt(): int	Returns next token as an int.
+nextLong(): long	Returns next token as a long.
+nextFloat(): float	Returns next token as a float.
+nextDouble(): double	Returns next token as a double.
+useDelimiter(pattern: String): Scanner	Sets this scanner's delimiting pattern.

File Input and Output

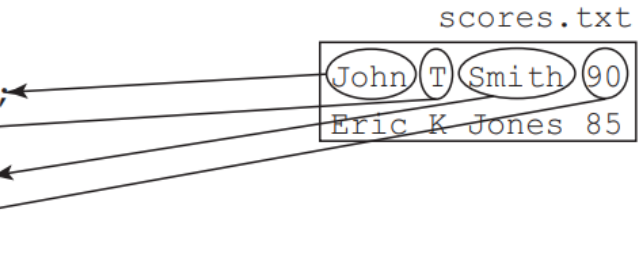
12.11.3 Reading Data Using Scanner

ReadData.java

```
1  import java.util.Scanner;
2
3  public class ReadData {
4    public static void main(String[] args) throws Exception {
5      // Create a File instance
6      java.io.File file = new java.io.File("scores.txt");
7
8      // Create a Scanner for the file
9      Scanner input = new Scanner(file);
10
11     // Read data from a file
12     while (input.hasNext()) {
13       String firstName = input.next();
14       String mi = input.next();
15       String lastName = input.next();
16       int score = input.nextInt();
17       System.out.println(
18         firstName +" " + mi + " " + lastName + " " + score);
19     }
20
21     // Close the file
22     input.close();
23   }
24 }
```

create a File

create a Scanner



has next?
read items

close file

File Input and Output

Closing Resources Automatically Using try-with-resources

Programmers often forget to close the file. JDK 7 provides the following try-with-resources syntax that **automatically closes the files**.

```
try (declare and create resources) {

        Use the resource to process the file;

}
```

File Input and Output

Closing Resources Automatically Using try-with-resources

WriteDataWithAutoClose.java

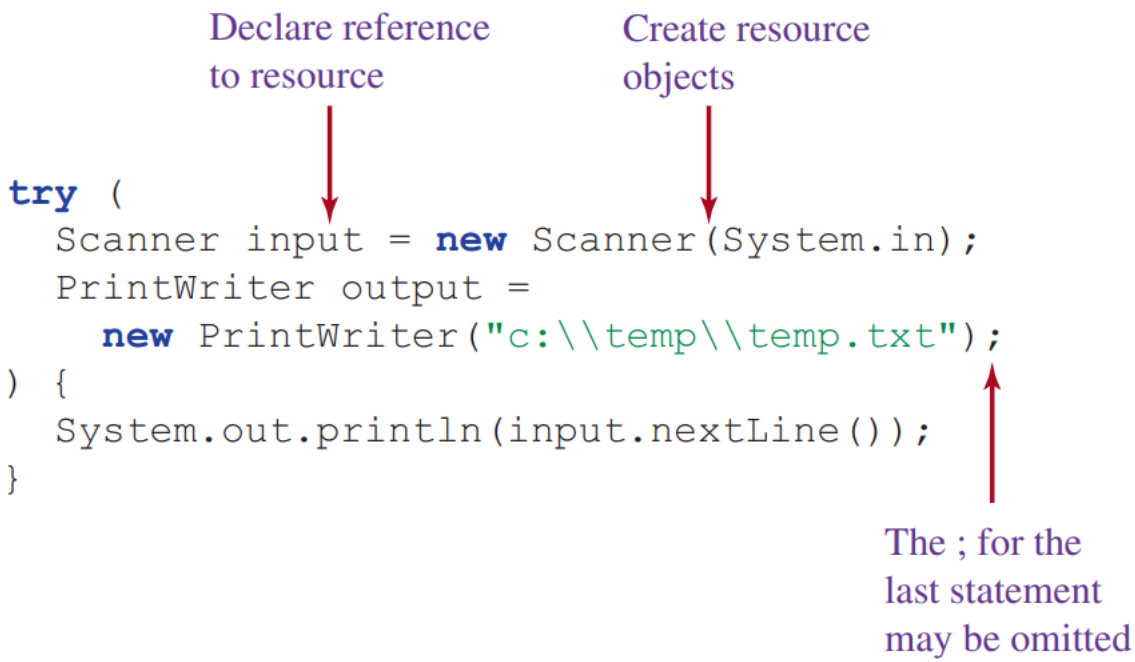
```
1  public class WriteDataWithAutoClose {
2    public static void main(String[] args) throws Exception {
3      java.io.File file = new java.io.File("scores.txt");
4      if (file.exists()) {
5        System.out.println("File already exists");
6        System.exit(0);
7      }
8
9      try (
10       // Create a file
11       java.io.PrintWriter output = new java.io.PrintWriter(file);
12     ) {
13       // Write formatted output to the file
14       output.print("John T Smith ");
15       output.println(90);
16       output.print("Eric K Jones ");
17       output.println(85);
18     }
19   }
20 }
```

declare/create resource

use the resource

File Input and Output

Closing Resources Automatically Using try-with-resources



CHAPTER 17

Binary I/O

Introduction

Java provides many classes for performing text I/O and binary I/O.

- Data stored in a text file is represented in human-readable form.
- Data stored in a binary file is represented in binary form.
- Binary files are designed to be read by programs.

  For example,
  - Java source programs are stored in text files and can be read by a text editor, but
  - Java classes are stored in binary files and are read by the JVM.

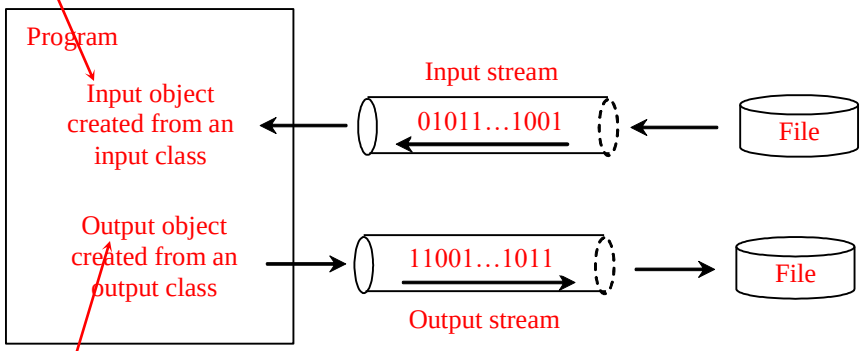
The advantage of binary files is that they are more efficient to process than text files.

How is I/O Handled in Java?

Text data are read using the Scanner class and written using the PrintWriter class.

Recall:

```
Scanner input = new Scanner(new File("temp.txt"));
System.out.println(input.nextLine());
```




```
PrintWriter output = new PrintWriter("temp.txt");
output.println(" Comp 2311 ");
output.close();
```

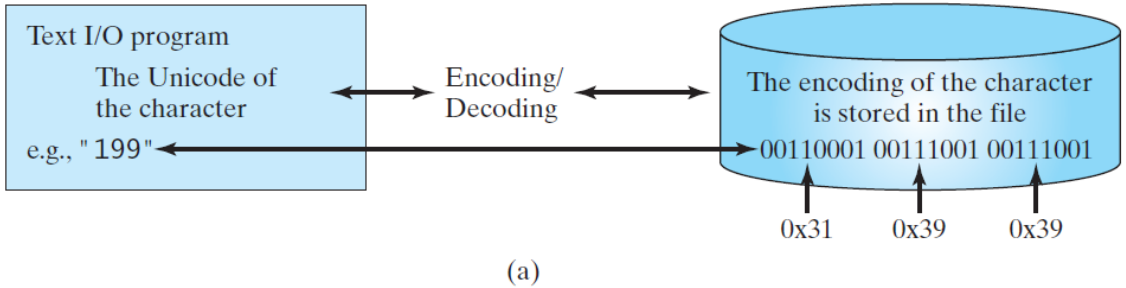
READING FROM

WRITING TO

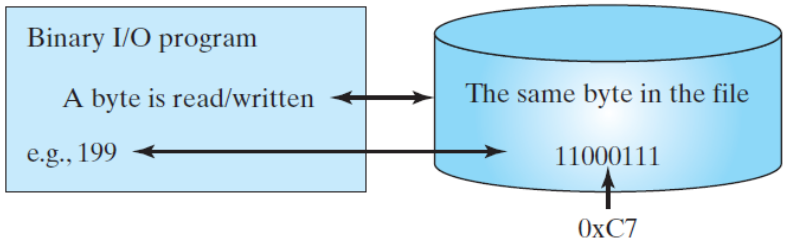
Text I/O vs. Binary I/O

Binary I/O does not involve encoding or decoding and thus is more efficient than text I/O.

- You can imagine that a text file consists of a sequence of characters and
- A binary file consists of a sequence of bits.



(a)



(b)

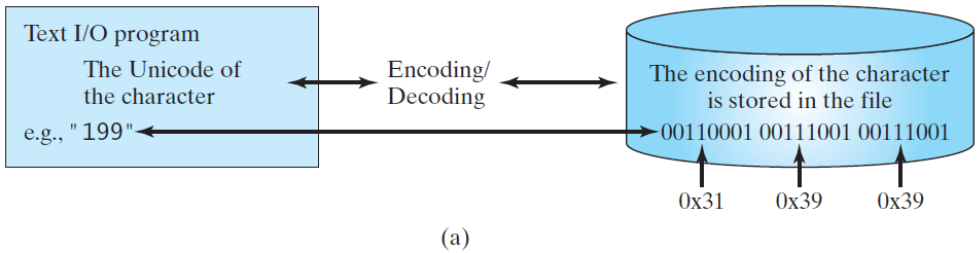
For example, the decimal integer 199 is stored as the sequence of three characters: '1', '9', '9' in a text file.

The same integer is stored as a byte-type value C7 in a binary file, because decimal 199 equals to hex C7 (Binary: 11000111).

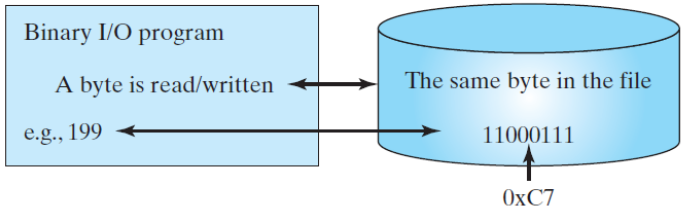
Binary I/O

- Text I/O requires encoding and decoding.
  - The JVM encodes the characters to their Unicode when writing out to a file and decodes the Unicode to a character when reading from a file.

- Binary I/O does not require conversions.
  - When you write a byte to a file, the original byte is copied into the file.
  - When you read a byte from a file, the exact byte in the file is returned.



(a)

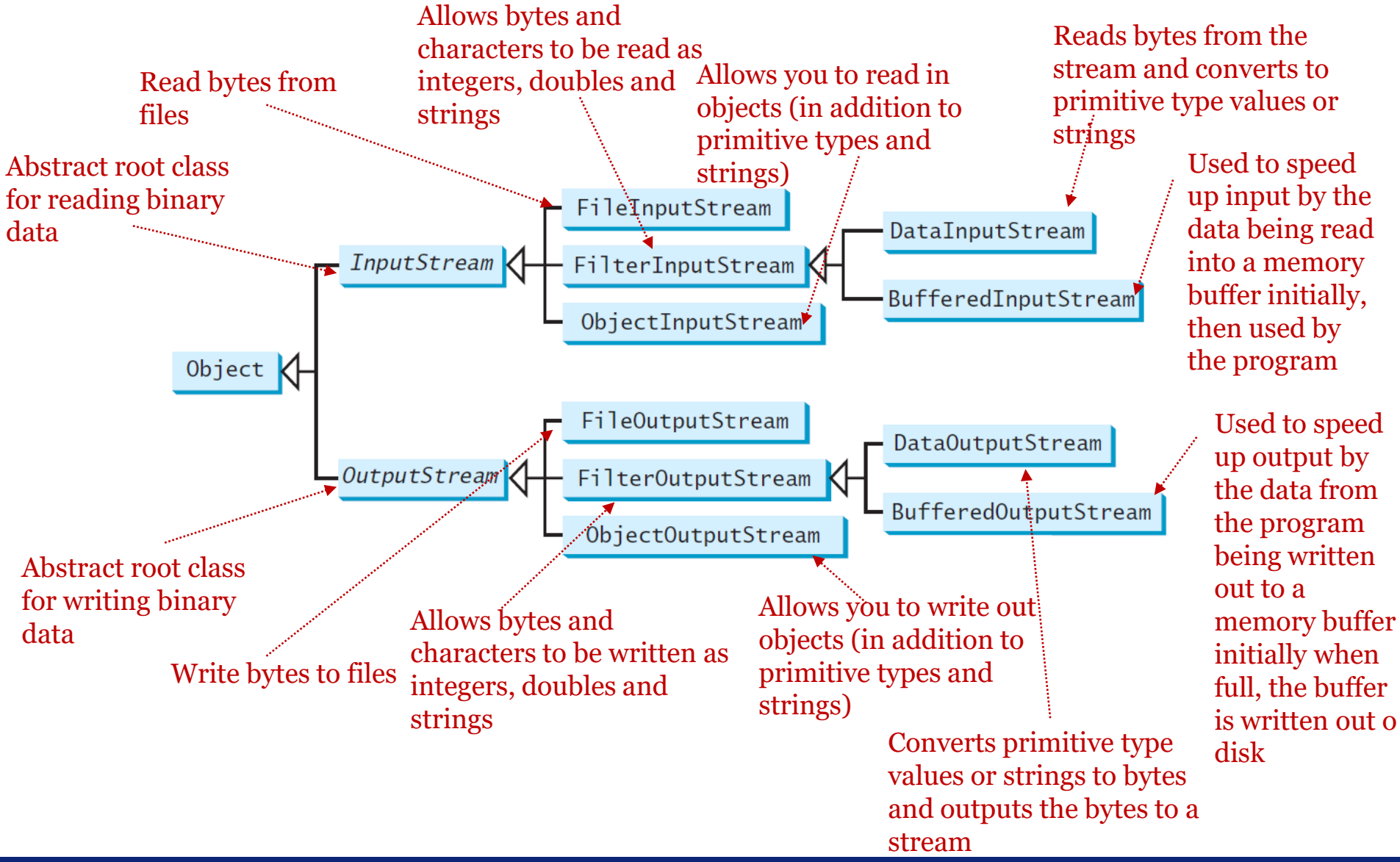


(b)

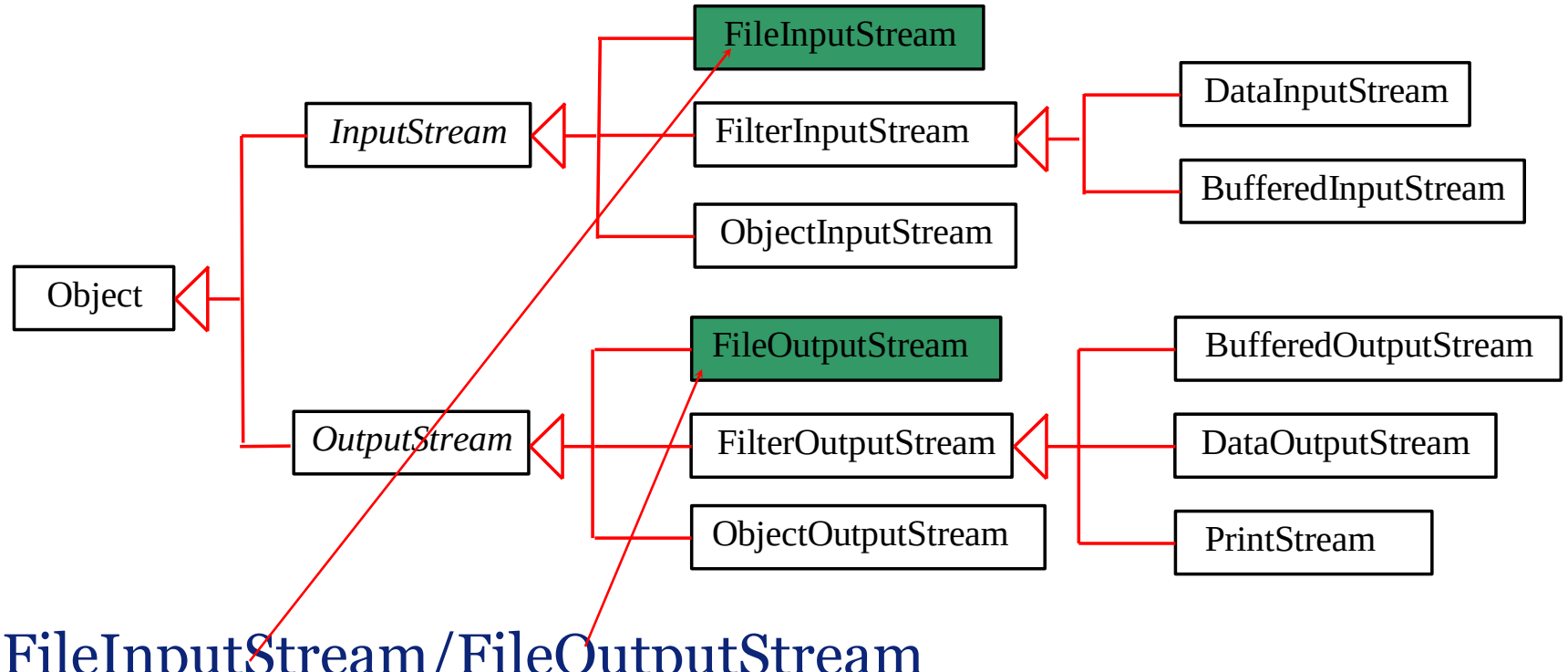
For example, the decimal integer 199 is stored as the sequence of three characters: '1', '9', '9' in a text file.

The same integer is stored as a byte-type value C7 in a binary file, because decimal 199 equals to hex C7 (Binary: 11000111).

Binary I/O Classes



FileInputStream/FileOutputStream



FileInputStream/FileOutputStream

- Associates a binary input/output stream with an external file.
- All the methods in FileInputStream/FileOuptputStream are inherited from its superclasses.

Binary I/O Classes

InputStream

Abstract root class for reading binary data

The value returned is a byte as an int type.

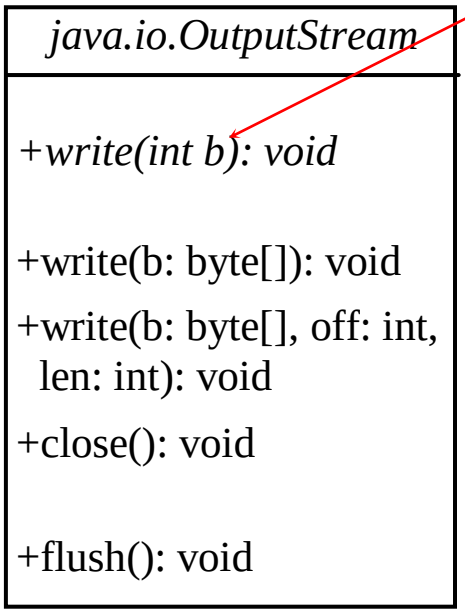
java.io.InputStream	
+read(): int	Reads the next byte of data from the input stream. The value byte is returned as an int value in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value –1 is returned.
+read(b: byte[]): int	Reads up to b.length bytes into array b from the input stream and returns the actual number of bytes read. Returns -1 at the end of the stream.
+read(b: byte[], off: int, len: int): int	Reads bytes from the input stream and stores into b[off], b[off+1], …, b[off+len-1]. The actual number of bytes read is returned. Returns -1 at the end of the stream.
+available(): int	Returns the number of bytes that can be read from the input stream.
+close(): void	Closes this input stream and releases any system resources associated with the stream.
+skip(n: long): long	Skips over and discards n bytes of data from this input stream. The actual number of bytes skipped is returned.
+markSupported(): boolean	Tests if this input stream supports the mark and reset methods.
+mark(readlimit: int): void	Marks the current position in this input stream.
+reset(): void	Repositions this stream to the position at the time the mark method was last called on this input stream.

Binary I/O Classes

OutputStream

Abstract root class for writing binary data

The value is a byte as an int type.



Writes the specified byte to this output stream. The parameter b is an int value. (byte)b is written to the output stream.

Writes all the bytes in array b to the output stream.

Writes b[off], b[off+1], …, b[off+len-1] into the output stream.

Closes this output stream and releases any system resources associated with the stream.

Flushes this output stream and forces any buffered output bytes to be written out.

FileInputStream Read bytes from files

To construct a FileInputStream, use the following constructors:

public FileInputStream(String filename)

public FileInputStream(File file)

A java.io.FileNotFoundException would occur if you attempt to create a FileInputStream with a nonexistent file.

FileOutputStream Write bytes to files

To construct a FileOutputStream, use the following constructors:

public FileOutputStream(String filename)
public FileOutputStream(File file)
public FileOutputStream(String filename, boolean append)
public FileOutputStream(File file, boolean append)

- If the file does not exist, a new file would be created. If the file already exists, the first two constructors would delete the current contents in the file.
- To retain the current content and append new data into the file, use the last two constructors by passing true to the append parameter.

Example

Write simple program that uses binary I/O to write 10 byte values from 1 to 10 to a file named temp.dat and read them back from the file

```
1  import java.io.*;
2
3  public class TestFileStream {
4    public static void main(String[] args) throws IOException {
5      try (
6        // Create an output stream to the file
7        FileOutputStream output = new FileOutputStream("temp.dat");
8      ) {
9        // Output values to the file
10       for (int i = 1; i <= 10; i++)
11         output.write(i);
12     }
13
14     try (
15       // Create an input stream for the file
16       FileInputStream input = new FileInputStream("temp.dat");
17     ) {
18       // Read values from the file
19       int value;
20       while ((value = input.read()) != -1)
21         System.out.print(value + " ");
22     }
23   }
24 }
```

Import needed

Creates output stream

Output to file

Create input stream

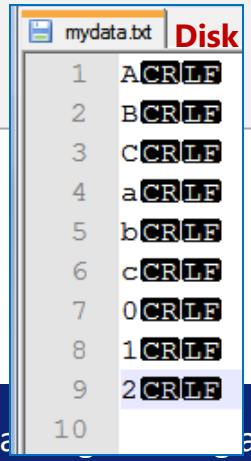
Read values from file

An input value of -1 signifies the end of file

Example: FileInputStream

FileInputStream is for reading/writing bytes from/to a disk file.

```
public static void main(String[] args) throws IOException {

    FileInputStream fis1 = new FileInputStream("c:\\temp\\mydata.txt");
    //alternatively you may also say...
    File file = new File("c:\\temp\\mydata.txt");
    FileInputStream fis2 = new FileInputStream( file );

    int avail = fis1.available();

    for(int i=0; i<avail; i++ ){
        int data = fis1.read();
        System.out.print(data  );
    }

}//main
```



13 **C**arriage **R**eturn
10 **L**ine **F**eed

available 27 **CONSOLE**

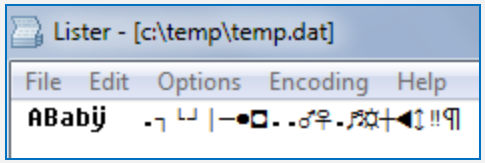
65 13 10 **66** 13 10 **67** 13 10 **97** 13 10 **98** 13 10 **99** 13 10 **48** 13 10 **49** 13 10 **50** 13 10

A B C a b c 0 1 2

Example: FileInputStream & FileOutputStream

```
import java.io.*;

public class TestFileStream {
  public static void main(String[] args) throws IOException {
    // Create an output stream to the file
    FileOutputStream output = new FileOutputStream("temp.dat");

    output.write('A');  output.write('B');
    output.write('a');  output.write('b');
    output.write(255);  output.write(256);

    // Output values to the file
    for (int i = 0; i <= 20; i++)
      output.write(i);

    // Close the output stream
    output.close();

    // Create an input stream for the file
    FileInputStream input = new FileInputStream("temp.dat");

    // Read values from the file
    int value;
    while ((value = input.read()) != -1)
       System.out.print(value + " ");

    // Close the output stream
    input.close();
  }
}
```

Disk File



CONSOLE

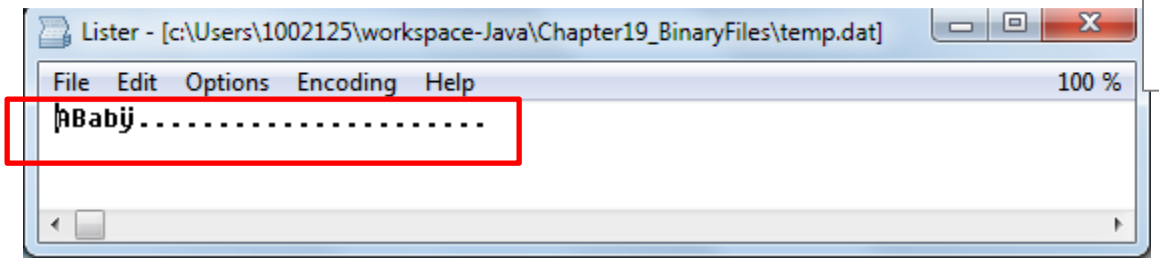
65 66 97 98 255 0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

A, B, a, b, 255, 256

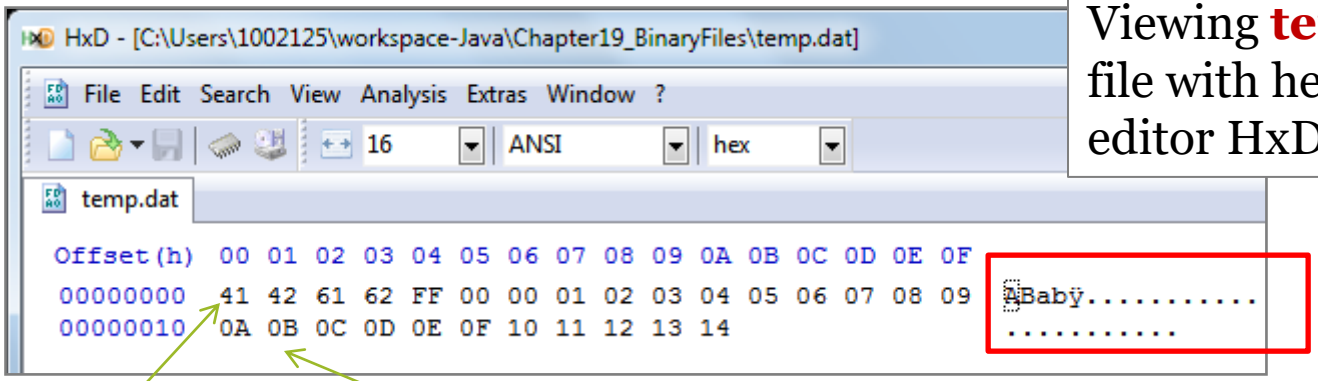
0, 1, 2, ... , 20

Example: FileInputStream & FileOutputStream

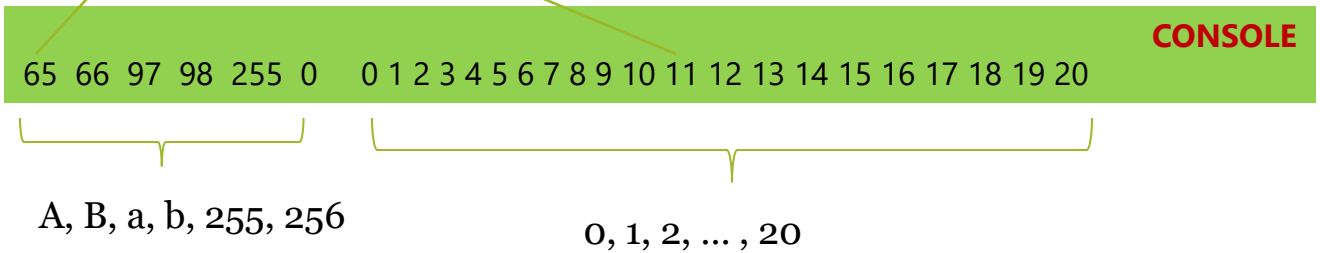
Continuation…



Viewing **temp.dat** file with MS-Lister



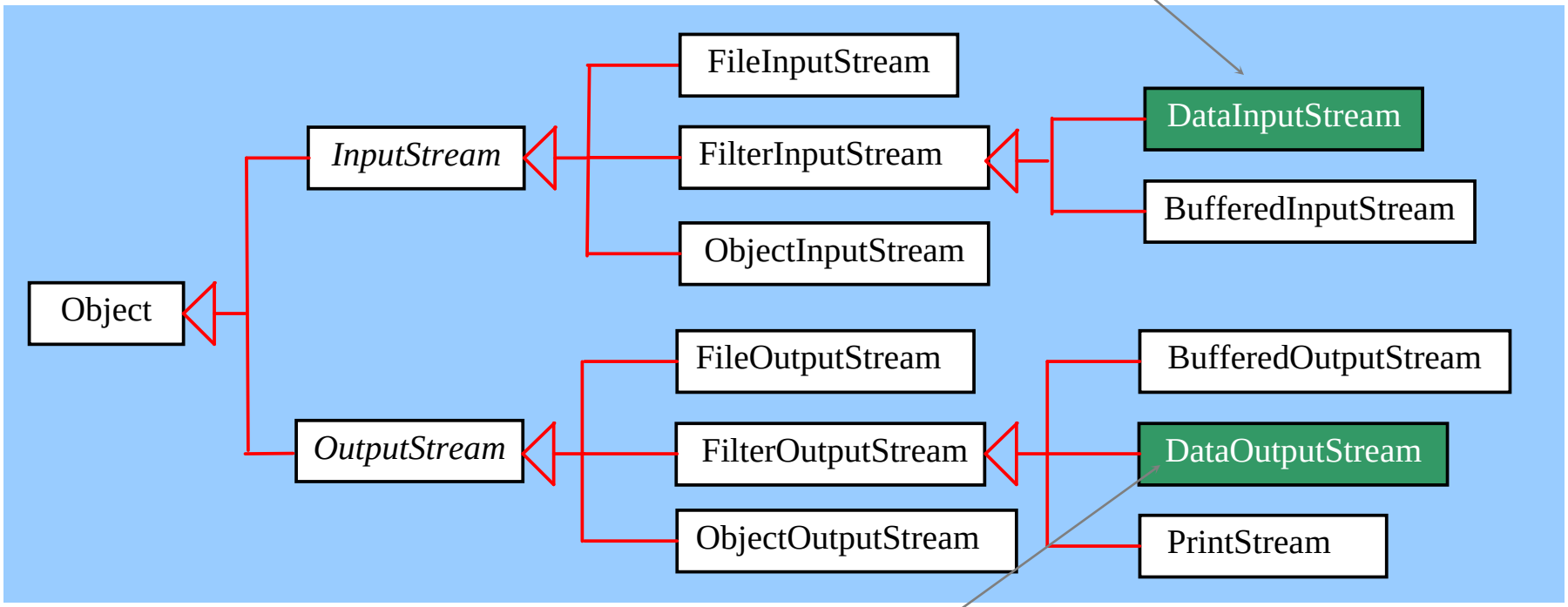
Viewing **temp.dat** file with hexadecimal editor HxD Hexedit.




```
Hex 41 =  Binary 1000 0001 = int 65   =  Char 'A'
            4    1
```

DataInputStream/DataOutputStream

DataInputStream reads bytes from the stream and converts them into appropriate primitive type values or strings.



DataOutputStream converts primitive type values or strings into bytes and output the bytes to the stream.

DataInputStream

`DataInputStream` reads bytes from the stream and converts them into appropriate primitive type values or strings.

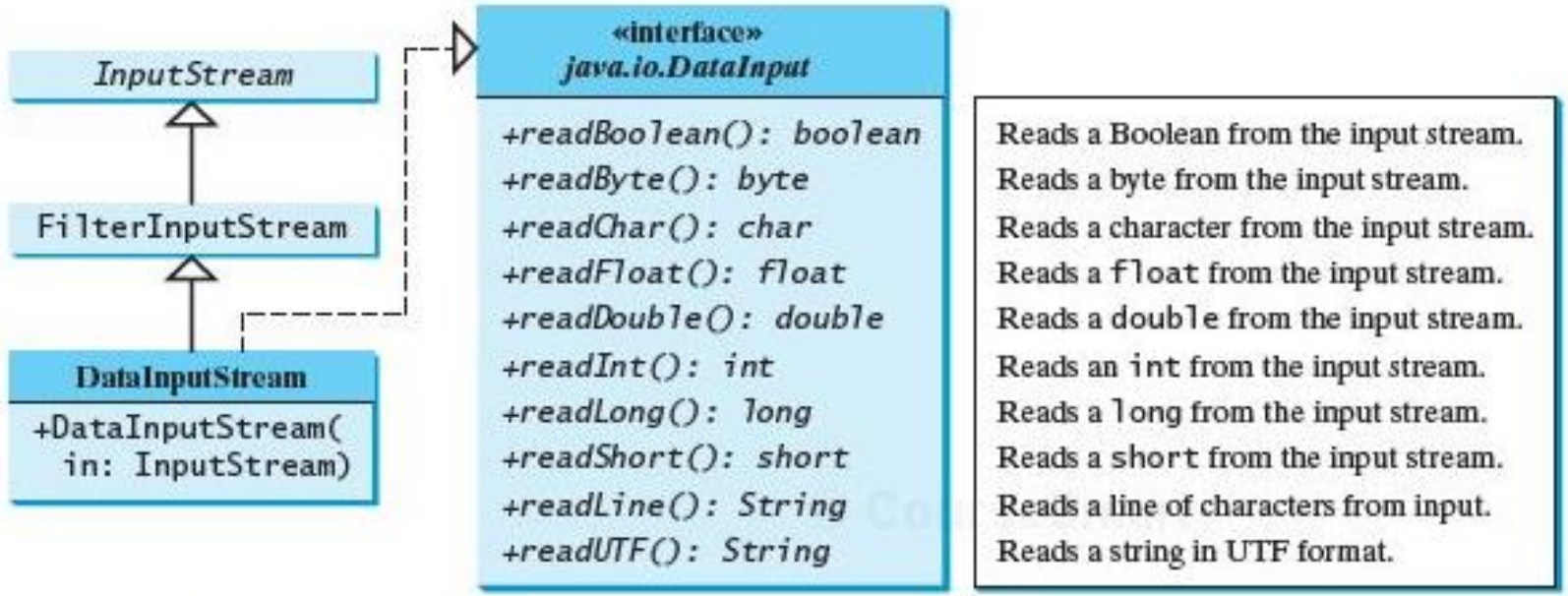


FIGURE 19.9 `DataInputStream` filters an input stream of bytes into primitive data type values and strings.

DataOutputStream

`DataOutputStream` converts primitive type values or strings into bytes and outputs the bytes to the stream.

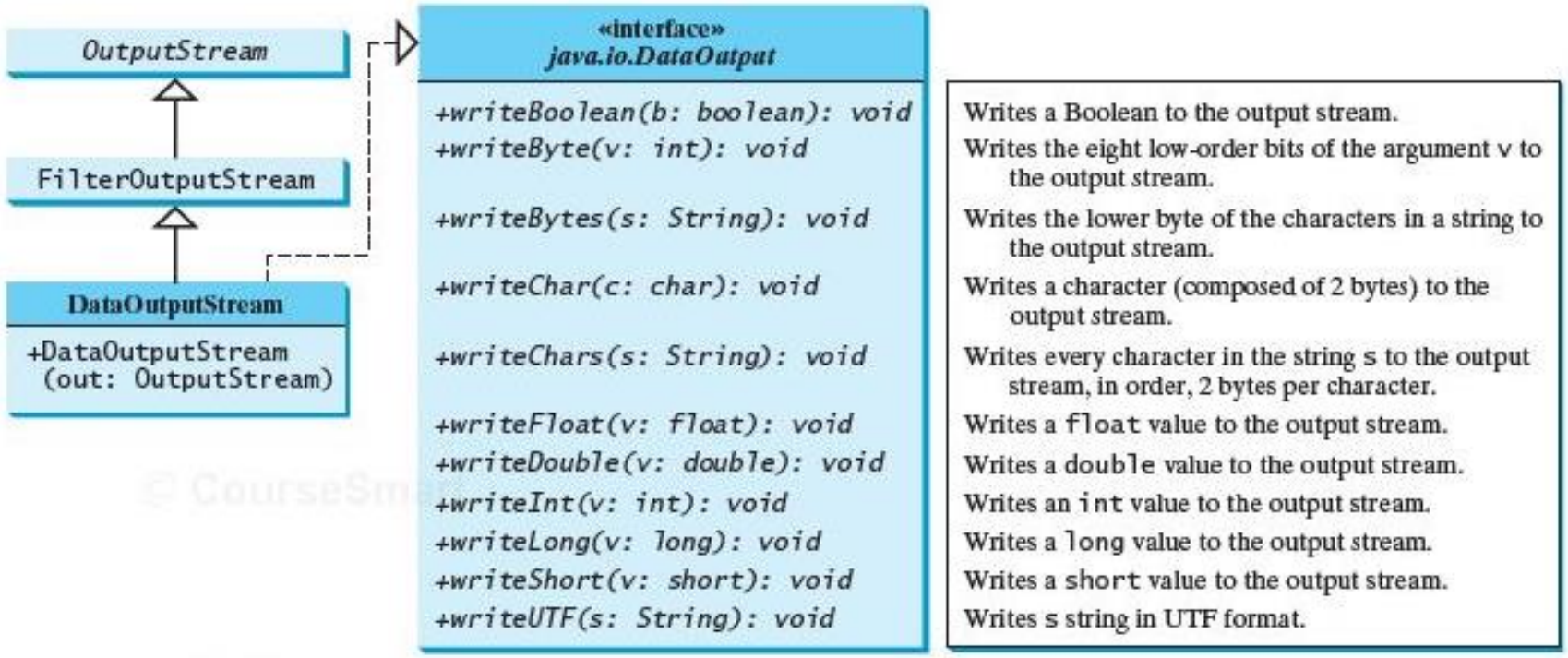


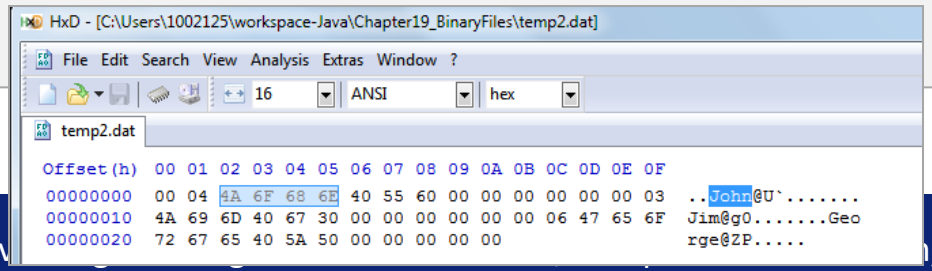
FIGURE 19.10 `DataOutputStream` enables you to write primitive data type values and strings into an output stream.

Example: Using DataInputStream/DataOutputStream

Self Study ☺

- Write then read student's name & score.

```
public class TestDataStream {
  public static void main(String[] args) throws IOException {
    // Create an output stream for file temp.dat
    DataOutputStream output = new DataOutputStream( new FileOutputStream("temp2.dat") );

    output.writeUTF("John");     output.writeDouble(85.5);
    output.writeUTF("Jim");      output.writeDouble(185.5);
    output.writeUTF("George");   output.writeDouble(105.25);

    // Close output stream
    output.close();

    // Create an input stream for file temp.dat
    DataInputStream input = new DataInputStream( new FileInputStream("temp2.dat") );

    // Read student test scores from the file
    System.out.println(input.readUTF() + " " + input.readDouble());
    System.out.println(input.readUTF() + " " + input.readDouble());
    System.out.println(input.readUTF() + " " + input.readDouble());
  }
}
```



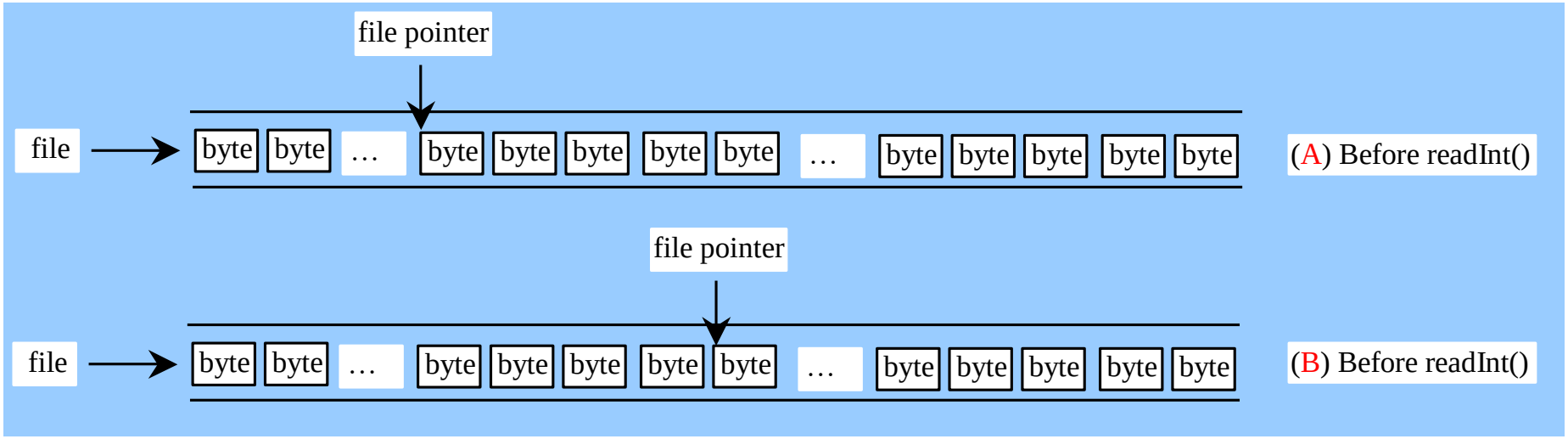

```
Console
John 85.5
Jim 185.5
George 105.25
```

The RandomAccessFile Class

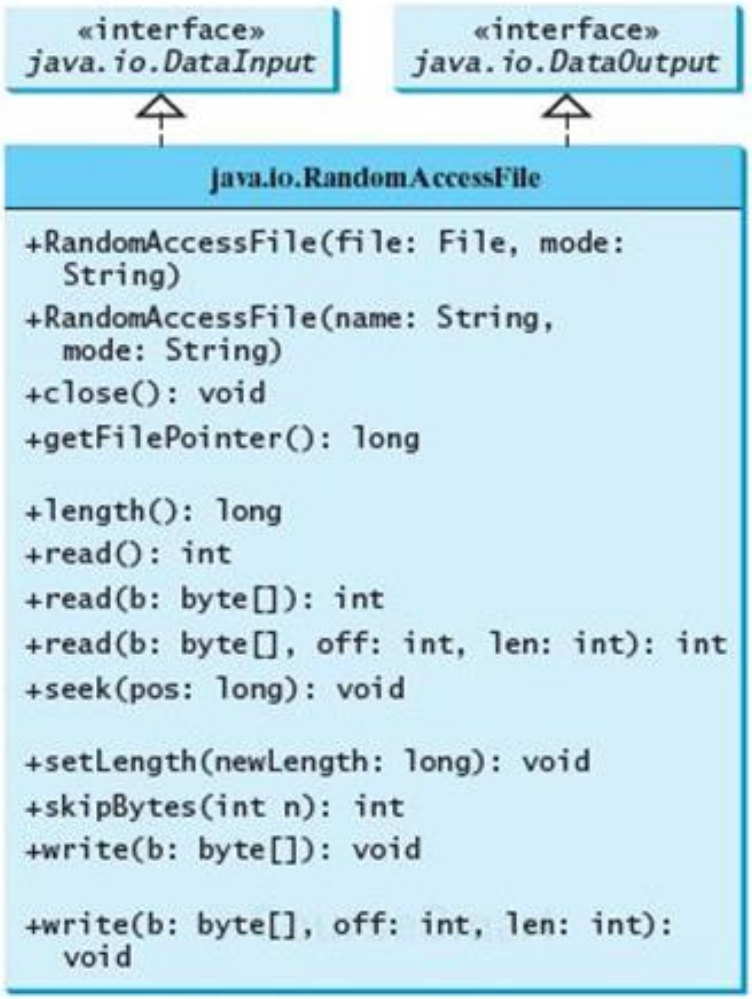
- All of the streams we have used so far are known as *read- only* or *write- only* streams.
- The external container of these streams are *sequential files* that cannot be updated without creating a new file.
- It is often necessary to change data in the files by *inserting*, *deleting*, or *re-writing* records.
- Java provides the **RandomAccessFile** class to allow a file to be read from and written to at *random locations*.

File Pointer

1. A random access file consists of a sequence of bytes.
2. There is a special marker called *file pointer* that is positioned at one of these bytes.
3. A read or write operation takes place at the location of the file pointer.
4. When a file is opened, the file pointer sets at the beginning of the file.



RandomAccessFile



`+RandomAccessFile(file: File, mode: String)`	Creates a `RandomAccessFile` stream with the specified `File` object and mode.
`+RandomAccessFile(name: String, mode: String)`	Creates a `RandomAccessFile` stream with the specified file name, string, and mode.
`+close(): void`	Closes the stream and releases the resource associated with it.
`+getFilePointer(): long`	Returns the offset, in bytes, from the beginning of the file to where the next `read` or `write` occurs.
`+length(): long`	Returns the length of this file.
`+read(): int`	Reads a byte of data from this file and returns –1 at the end of stream.
`+read(b: byte[]): int`	Reads up to `b.length` bytes of data from this file into an array of bytes.
`+read(b: byte[], off: int, len: int): int`	Reads up to `len` bytes of data from this file into an array of bytes.
`+seek(pos: long): void`	Sets the offset (in bytes specified in `pos`) from the beginning of the stream to where the next `read` or `write` occurs.
`+setLength(newLength: long): void`	Sets a new length for this file.
`+skipBytes(int n): int`	Skips over `n` bytes of input.
`+write(b: byte[]): void`	Writes `b.length` bytes from the specified byte array to this file, starting at the current file pointer.
`+write(b: byte[], off: int, len: int): void`	Writes `len` bytes from the specified byte array, starting at offset `off`, to this file.

FIGURE 19.16 `RandomAccessFile` implements the `DataInput` and `DataOutput` interfaces with additional methods to support random access.

`RandomAccessFile` Methods

- `void seek(long pos) throws IOException;`

  Sets the offset from the beginning of the `RandomAccessFile` stream to where the next read or write occurs.

- `long getFilePointer() throws IOException;`

  Returns the current offset, in bytes, from the beginning of the file to where the next read or write occurs.

`RandomAccessFile` Methods, cont.

- `long length()IOException`

  Returns the length of the file.

- `final void writeChar(int v) throws IOException`

  Writes a character to the file as a two-byte Unicode, with the high byte written first.

- `final void writeChars(String s)throws IOException`

  Writes a string to the file as a sequence of characters.

RandomAccessFile Constructor

- `// allows read and write`

```
RandomAccessFile raf =
 new RandomAccessFile( "test.dat", "rw" );
```

- `// read only`

```
RandomAccessFile raf =
 new RandomAccessFile( "test.dat", "r" );
```

Example: A Simple RandomAccessFile

```
public class TestRandomAccessFile {
  public static void main(String[] args) throws IOException {
    // Create a random access file
    RandomAccessFile inout = new RandomAccessFile("inout.dat", "rw");

    // Clear the file to destroy the old contents if exists
    inout.setLength(0);

    // Write new integers to the file
    for (int i = 0; i < 200; i++)
      inout.writeInt(i);

    // Display the current length of the file
    System.out.println("Current file length is " + inout.length());

    // Retrieve the first number
    inout.seek(0); // Move the file pointer to the beginning
    System.out.println("The first number is " + inout.readInt());

    // Retrieve the second number
    inout.seek(1 * 4); // Move the file pointer to the second number
    System.out.println("The second number is " + inout.readInt());

    // Retrieve the tenth number
    inout.seek(9 * 4); // Move the file pointer to the tenth number
    System.out.println("The tenth number is " + inout.readInt());
```

Console

```
Current file length is 800
The first number is 0
The second number is 1
The tenth number is 9
```

Example: A Simple RandomAccessFile

```
    // Modify the eleventh number
    inout.writeInt(555);

    // Append a new number
    inout.seek(inout.length()); // Move the file pointer to the end
    inout.writeInt(999);

    // Display the new length
    System.out.println("The new length is " + inout.length());

    // Retrieve the new eleventh number
    inout.seek(10 * 4); // Move the file pointer to the eleventh number
    System.out.println("The eleventh number is " + inout.readInt());

    inout.close();
  }
}
```

Console

```
Current file length is 800
The first number is 0
The second number is 1
The tenth number is 9
The new length is 804
The eleventh number is 555
```

Review Error Handling Binary I/O

Dr. Abdallah Karakra |Comp 2311 | Masri504

5/24/2023