BIRZEIT UNIVERSITY

Exception Handling and I/O

By: Mamoun Nawahdah (Ph.D.)

2022/2023

Exception Handling

❖ Exception handling technique enables a method to **throw** an exception **to its caller**.

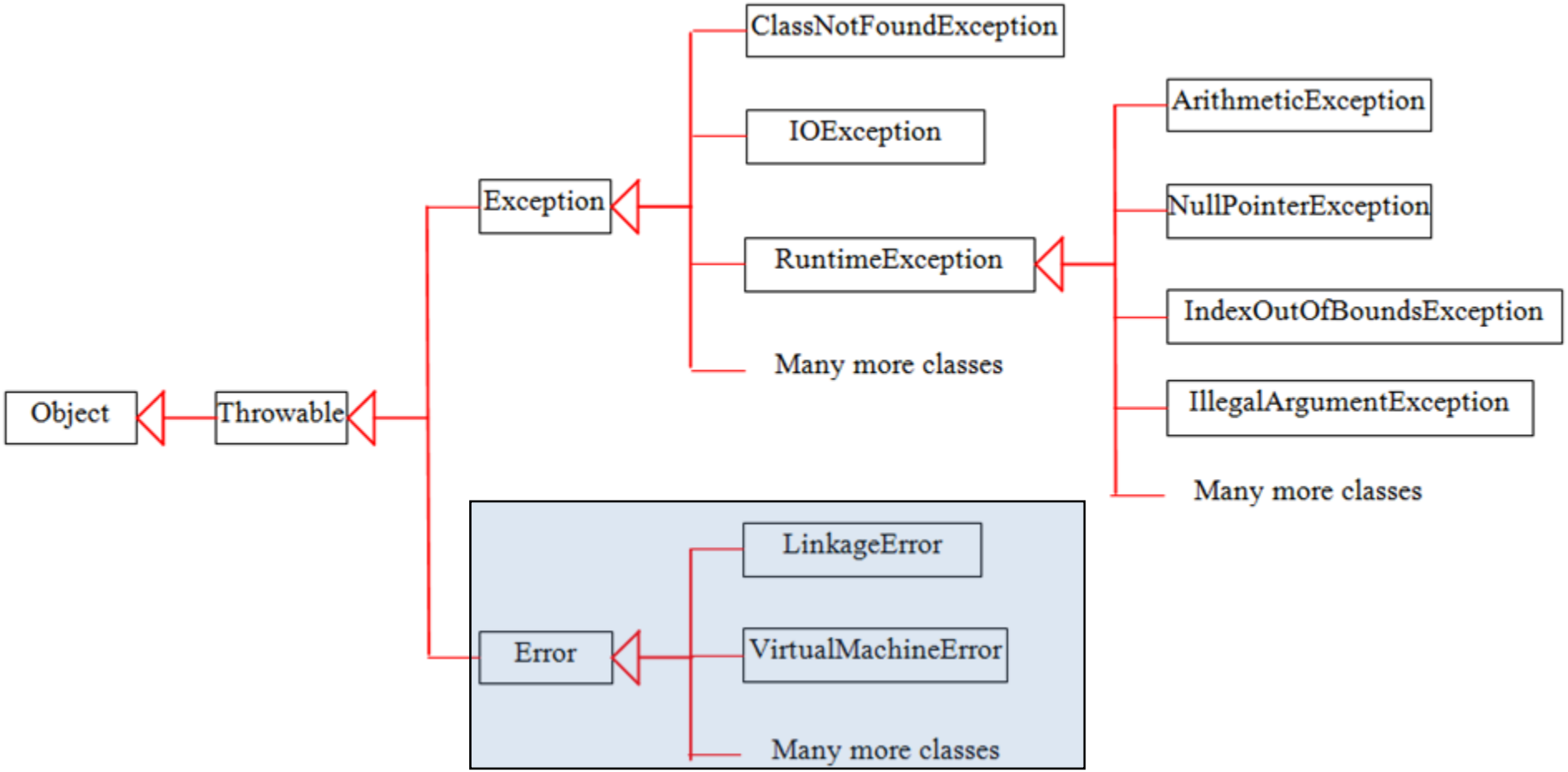
❖ Without this capability, a method must handle the exception or terminate the program.

ex·cep·tion *noun* \ik-'sep-shən\

: someone or something that is different from others : someone or something that is not included

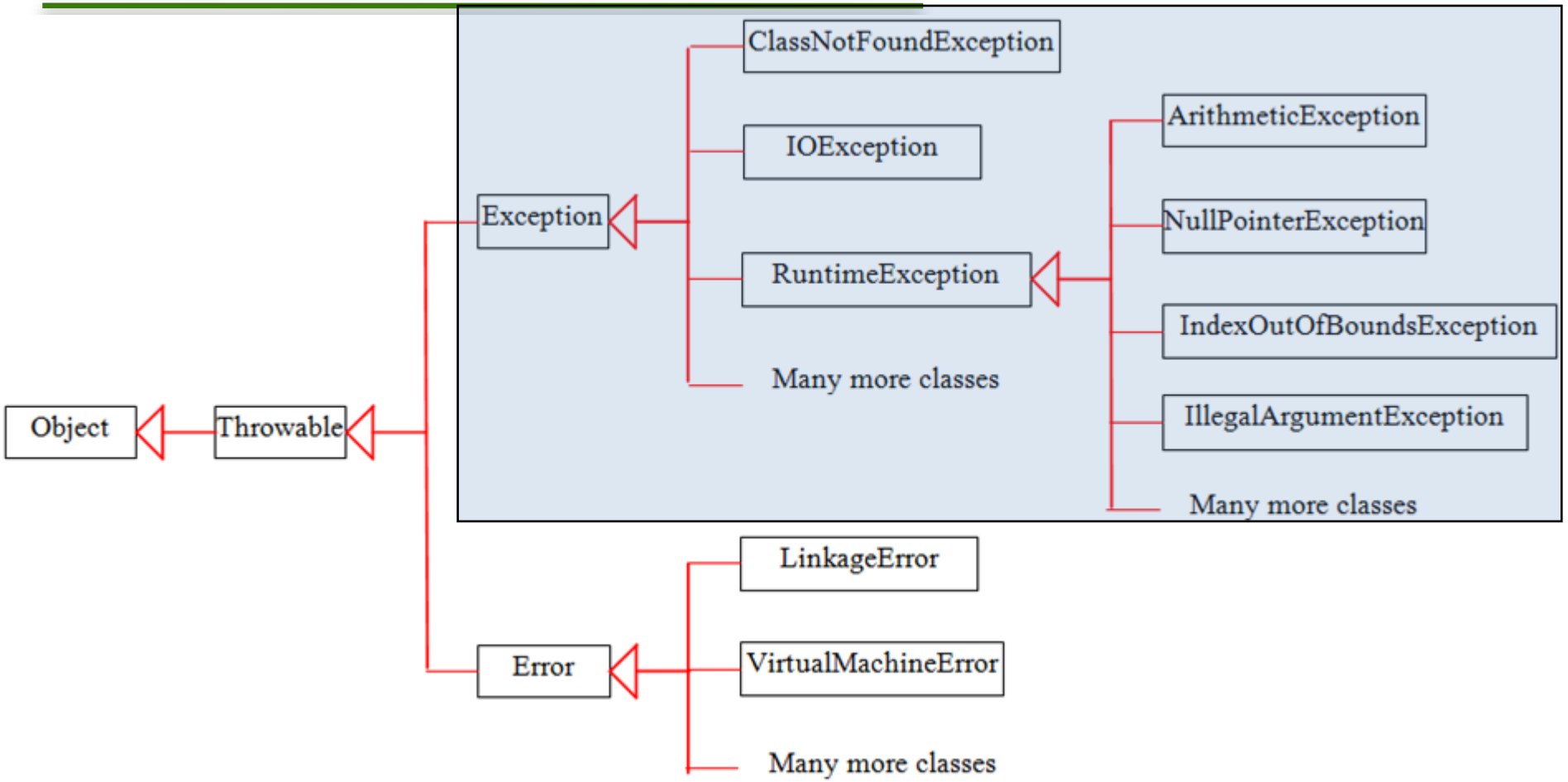
: a case where a rule does not apply

System Errors



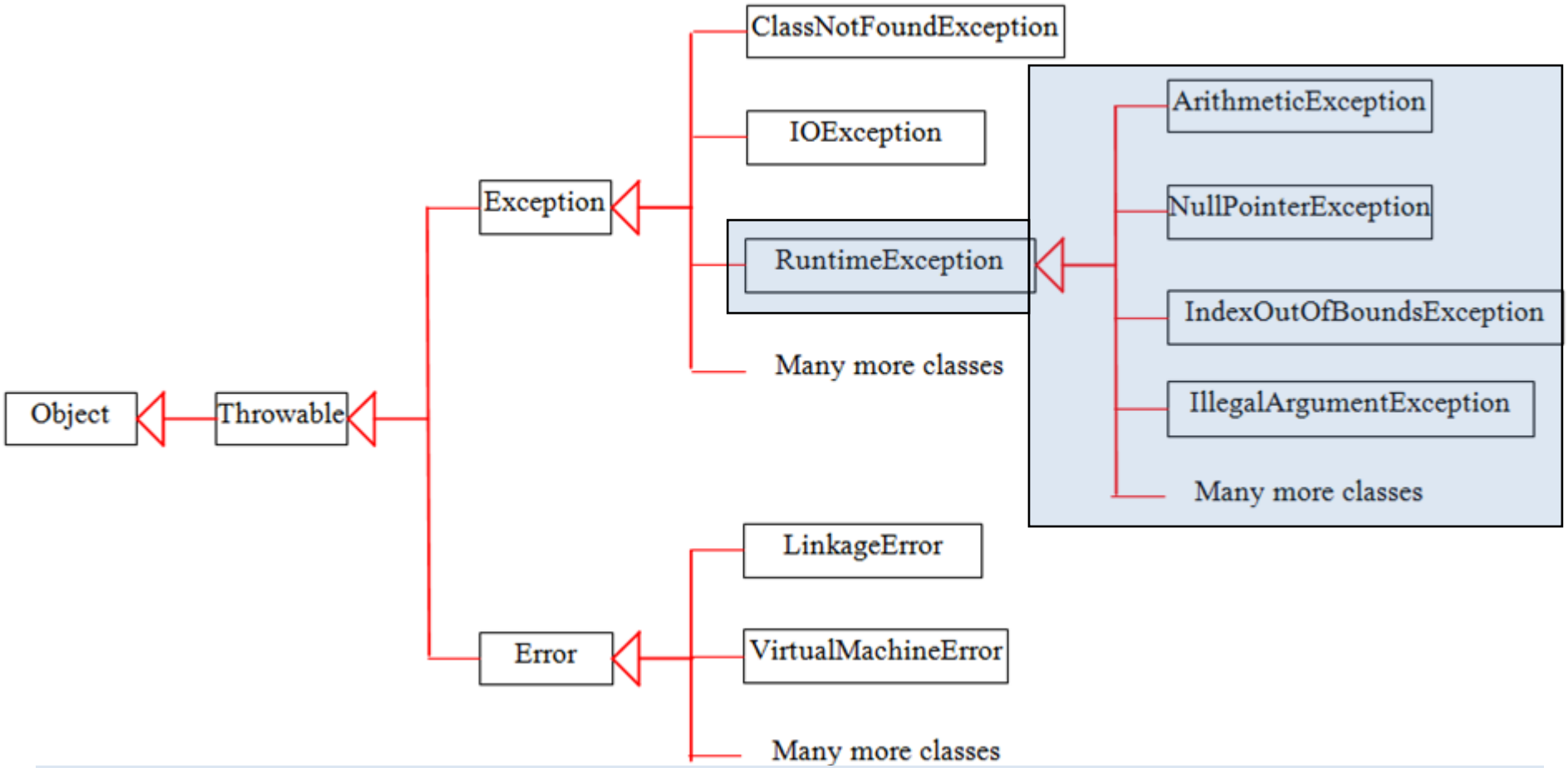
System errors are thrown by **JVM** and represented in the **Error** class. The Error class describes internal system errors.

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.

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.

Declaring, Throwing, and Catching Exceptions

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

declare exception

throw exception

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

catch exception

Declaring Exceptions

- Every method **must** state the types of checked exceptions it might **throw**.
- This is known as **declaring exceptions.**

```
public void x()  throws IOException

public void y()  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**.

```
throw new IOException();

IOException ex = new IOException();
throw ex;
```

Throwing Exceptions Example

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

Catching Exceptions

```
try {
    statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
    handler for exception1;
}
catch (Exception2 exVar2) {
    handler for exception2;
}
...
catch (ExceptionN exVar3) {
    handler for exceptionN;
}
```

The order in which exceptions are specified in catch blocks is important.

JDK 7 multicatch

You can use the new JDK 7 multicatch feature to simplify coding for the exceptions with the same handling code. The syntax is:

```
try {
    statements;  // Statements that may throw exceptions
}
catch (Exception1 | Exception2 |ExceptionN ex) {
    // handler exception;
}
```

Catch or Declare Checked Exceptions

Suppose **p2** is defined as follow:

```
void p2() throws IOException {
  if (a file does not exist) {
     throw new IOException("File does not exist");
  }

  ...
}
```

Catch or Declare Checked Exceptions

- Java forces you to deal with checked exceptions.
  - You must invoke it in a **try-catch** block **or**
  - declare to **throw** the exception in the calling method.
- For example, suppose that method **p1** invokes method **p2**, you have to write the code as follow:

```
void p1() {
  try {
    p2();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a)

```
void p1() throws IOException {

  p2();

}
```

(b)

```
1  public class CircleWithException {
2    /** The radius of the circle */
3    private double radius;
4
5    /** The number of the objects created */
6    private static int numberOfObjects = 0;
7
8    /** Construct a circle with radius 1 */
9    public CircleWithException() {
10     this(1.0);
11   }
12
13   /** Construct a circle with a specified radius */
14   public CircleWithException(double newRadius) {
15     setRadius(newRadius);
16     numberOfObjects++;
17   }
18
19   /** Return radius */
20   public double getRadius() {
21     return radius;
22   }
```

```
24    /** Set a new radius */
25    public void setRadius(double newRadius)
26        throws IllegalArgumentException {
27      if (newRadius >= 0)
28        radius =  newRadius;
29      else
30        throw new IllegalArgumentException(
31          "Radius cannot be negative");
32    }
33
34    /** Return numberOfObjects */
35    public static int getNumberOfObjects() {
36      return numberOfObjects;
37    }
38
39    /** Return the area of this circle */
40    public double findArea() {
41      return radius * radius * 3.14159;
42    }
43  }
```

```
1  public class TestCircleWithException {
2    public static void main(String[] args) {
3      try {
4        CircleWithException c1 = new CircleWithException(5);
5        CircleWithException c2 = new CircleWithException(-5);
6        CircleWithException c3 = new CircleWithException(0);
7      }
8      catch (IllegalArgumentException ex) {
9        System.out.println(ex);
10     }
11
12     System.out.println("Number of objects created: " +
13       CircleWithException.getNumberOfObjects());
14   }
15 }
```

Rethrowing Exceptions

```
try {
    statements;
}
catch(TheException ex) {
    perform operations before exits;
    throw ex;
}
```

The finally Clause

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

Note: The catch block may be omitted when the finally clause is used

Trace a Program Execution

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

Next statement;
```

Suppose no exceptions in the statements

Trace a Program Execution

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

Next statement;
```

The final block is always executed

Trace a Program Execution

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

Next statement;
```

Next statement in the method is executed

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

Trace a Program Execution

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

Next statement;
```

The exception is handled.

Trace a Program Execution

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

Next statement;
```

The final block is always executed.

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.

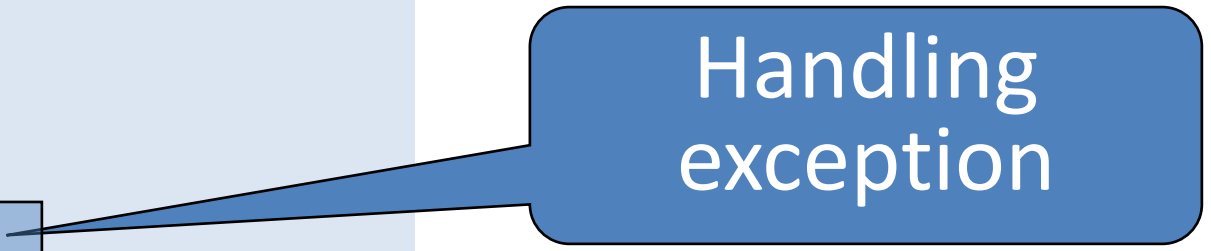
Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

statement2 throws an exception of type Exception2.

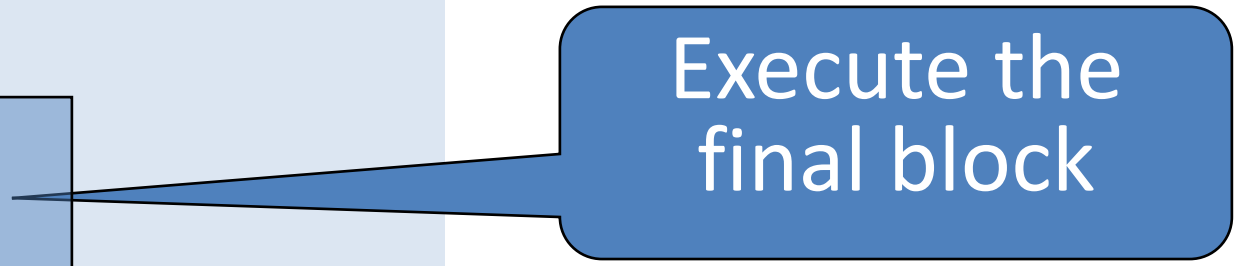
Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```



Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```



Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Rethrow the exception and control is transferred to the caller

Cautions When Using Exceptions

- 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*).
- 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 broadcasting the errors to the calling methods.

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.

When to Use Exceptions

- You should use it to deal with **unexpected** error conditions.

Caution!

❖ Do not use exception 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");
}
```

❖ is better to be replaced by:

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

Defining Custom Exception

- 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** class.

Custom Exception Class Example

```
1  public class InvalidRadiusException extends Exception {
2    private double radius;
3
4    /** Construct an exception */
5    public InvalidRadiusException(double radius) {
6      super("Invalid radius " + radius);
7      this.radius = radius;
8    }
9
10   /** Return the radius */
11   public double getRadius() {
12     return radius;
13   }
14 }
```

```
/** Set a new radius */
public void setRadius(double newRadius)
    throws InvalidRadiusException {
  if (newRadius >= 0)
    radius =  newRadius;
  else
    throw new InvalidRadiusException(newRadius);
}
```

The File Class

- The **File** class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion.
- The filename is a string.
- The **File** class is a wrapper class for the file name and its directory path.

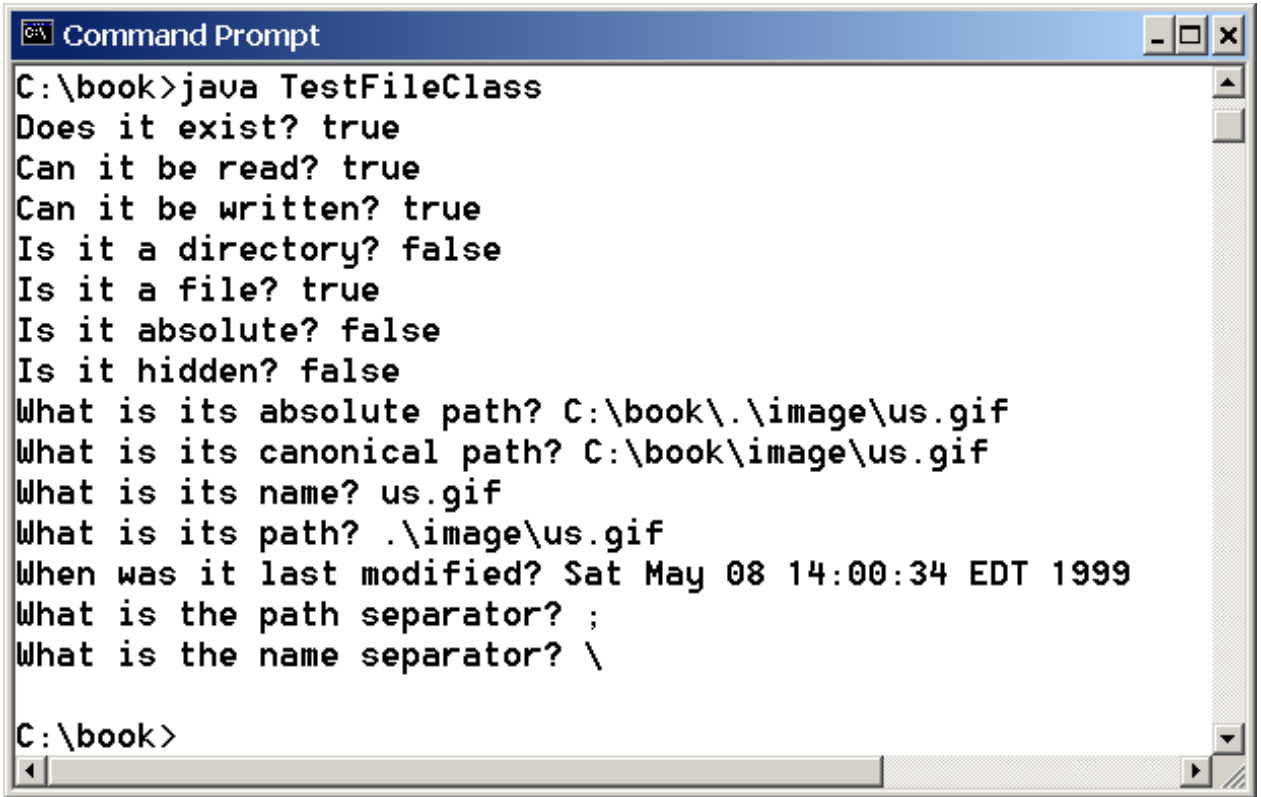
File class

java.io.File	
+File(pathname: String)	Creates a File object for the specified path name. The path name may be a directory or a file.
+File(parent: String, child: String)	Creates a File object for the child under the directory parent. The child may be a file name or a subdirectory.
+File(parent: File, child: String)	Creates a File object for the child under the directory parent. The parent is a File object. In the preceding constructor, the parent is a string.
+exists(): boolean	Returns true if the file or the directory represented by the File object exists.
+canRead(): boolean	Returns true if the file represented by the File object exists and can be read.
+canWrite(): boolean	Returns true if the file represented by the File object exists and can be written.
+isDirectory(): boolean	Returns true if the File object represents a directory.
+isFile(): boolean	Returns true if the File object represents a file.
+isAbsolute(): boolean	Returns true if the File object is created using an absolute path name.
+isHidden(): boolean	Returns true if the file represented in the File object is hidden. The exact definition of *hidden* is system-dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period(.) character.

The directory separator for Windows is a backslash (\). The backslash is a special character in Java and should be written as \\ in a string literal

Explore File Properties

- Write a program that demonstrates how to create files and use the methods in the File class to obtain their properties. The following figures show a sample run of the program:



Text File Input and Output

- In order to perform I/O, you need to create objects using appropriate Java I/O classes.
- There are two types of files: **text** and **binary**.
- Text files are essentially characters on disk.
- The objects contain the methods for reading/writing text data from/to a file.
- This section introduces how to read/write strings and numeric values from/to a text file using the **Scanner** and **PrintWriter** classes.

PrintWriter class

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 §3.6, "Formatting Console Output and Strings."

The close() method must be used to close the file. If this method is not invoked, the data may not be saved properly in the file.

Scanner class

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.

Try-with-resources

Programmers often forget to close the file.

JDK 7 provides the followings new try-with-resources syntax that automatically closes the files.

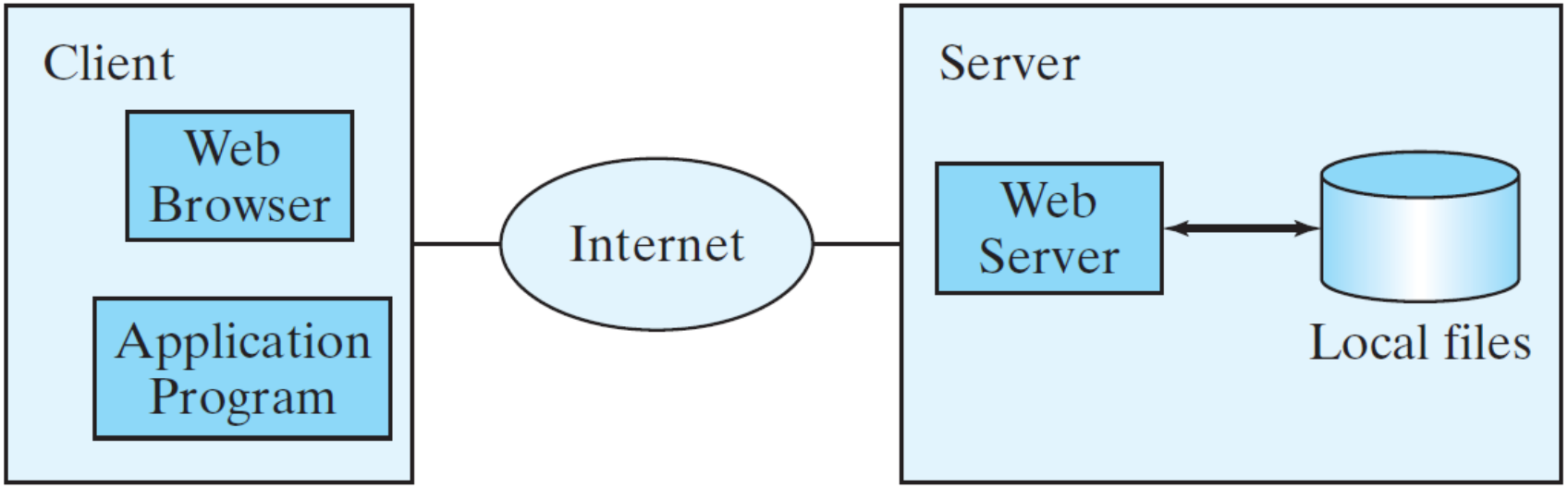
```
import java.io.*;
public class WriteDataWithAutoClose {
  public static void main(String[] args) throws Exception {
    File file = new File("data.txt");
    try ( PrintWriter output = new PrintWriter(file); ) {
        output.println("Mamoun Nawahdah");
        output.println("Birzeit University");
     }
   }
}
```

Problem: Replacing Text

- Write a class named **ReplaceText** that replaces a string in a text file with a new string. The filename and strings are passed as command-line arguments as follows:
  - **java ReplaceText sourceFile targetFile oldString newString**

Reading Data from the Web

Just like you can read data from a file on your computer, you can read data from a file on the Web.



Reading Data from the Web

```
URL url = new
URL("www.google.com/index.html");
```

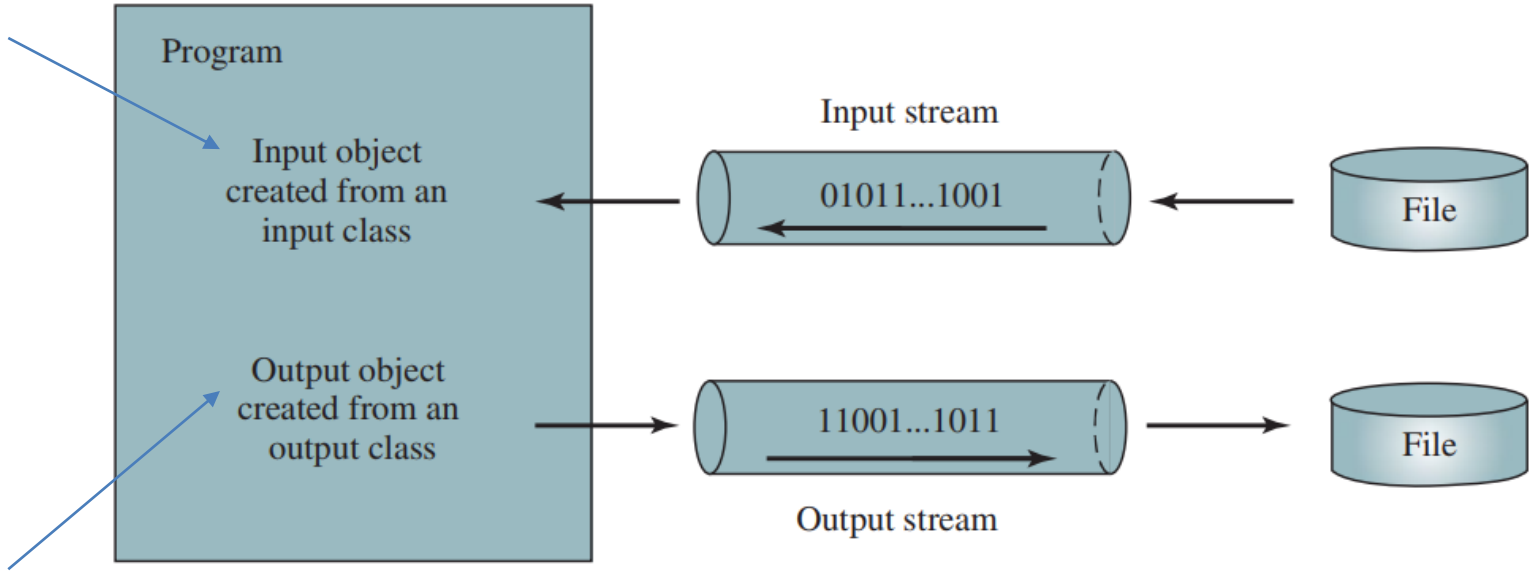
- After a **URL** object is created, you can use the **openStream()** method defined in the **URL** class to open an input stream and use this stream to create a **Scanner** object as follows:

```
Scanner input = new
Scanner(url.openStream());
```

How is I/O Handled in Java?

In order to perform I/O, you need to create objects using appropriate Java I/O classes.

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



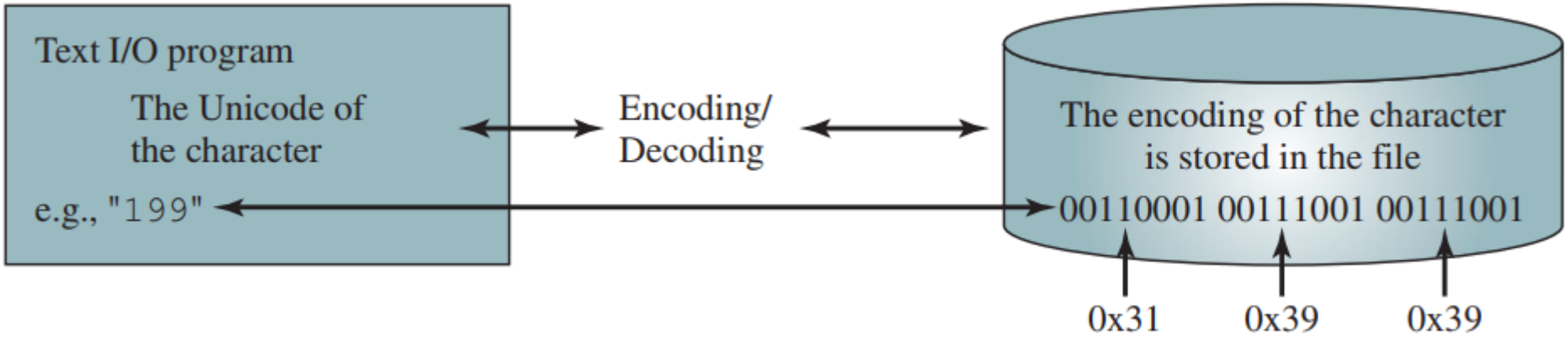

```
PrintWriter output = new PrintWriter("temp.txt");
output.println("Mamoun Nawahdah");
output.close();
```

Text File vs. Binary File

- Data stored in a text file are represented in **human-readable** form.
- Data stored in a binary file are represented in **binary** form.
- For example, the *Java source* programs are stored in text files and can be read by a text editor, but the *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.

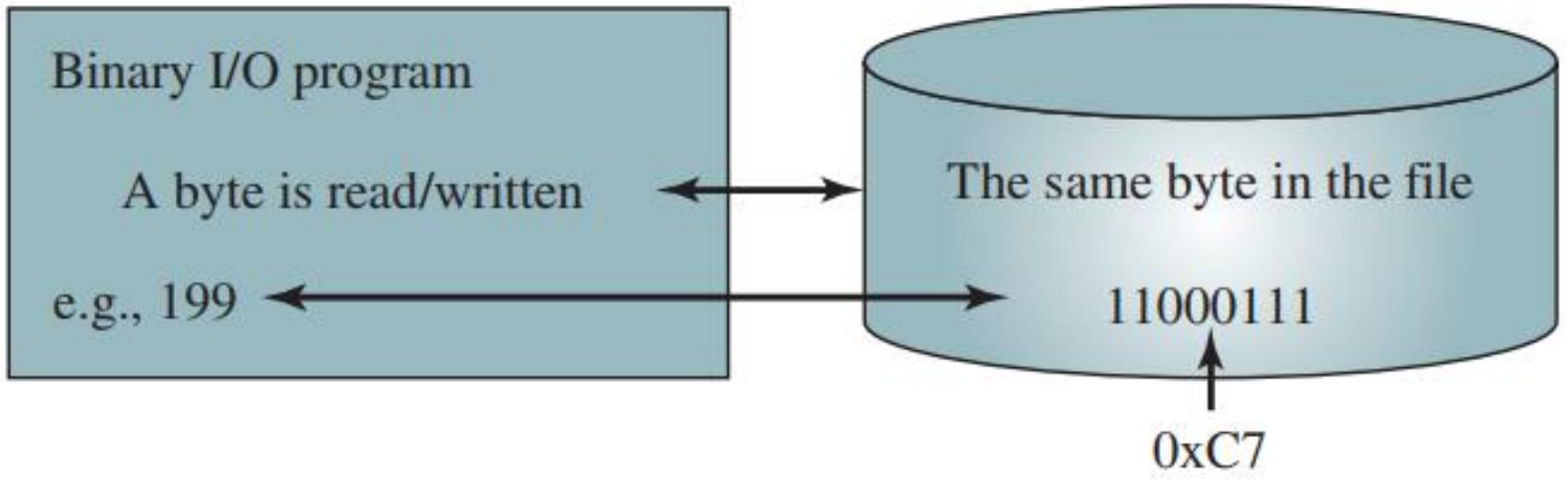
Text I/O

- Text I/O requires **encoding** and **decoding**.
- The JVM converts a Unicode to a file specific encoding when writing a character and coverts a file specific encoding to a Unicode when reading a character.

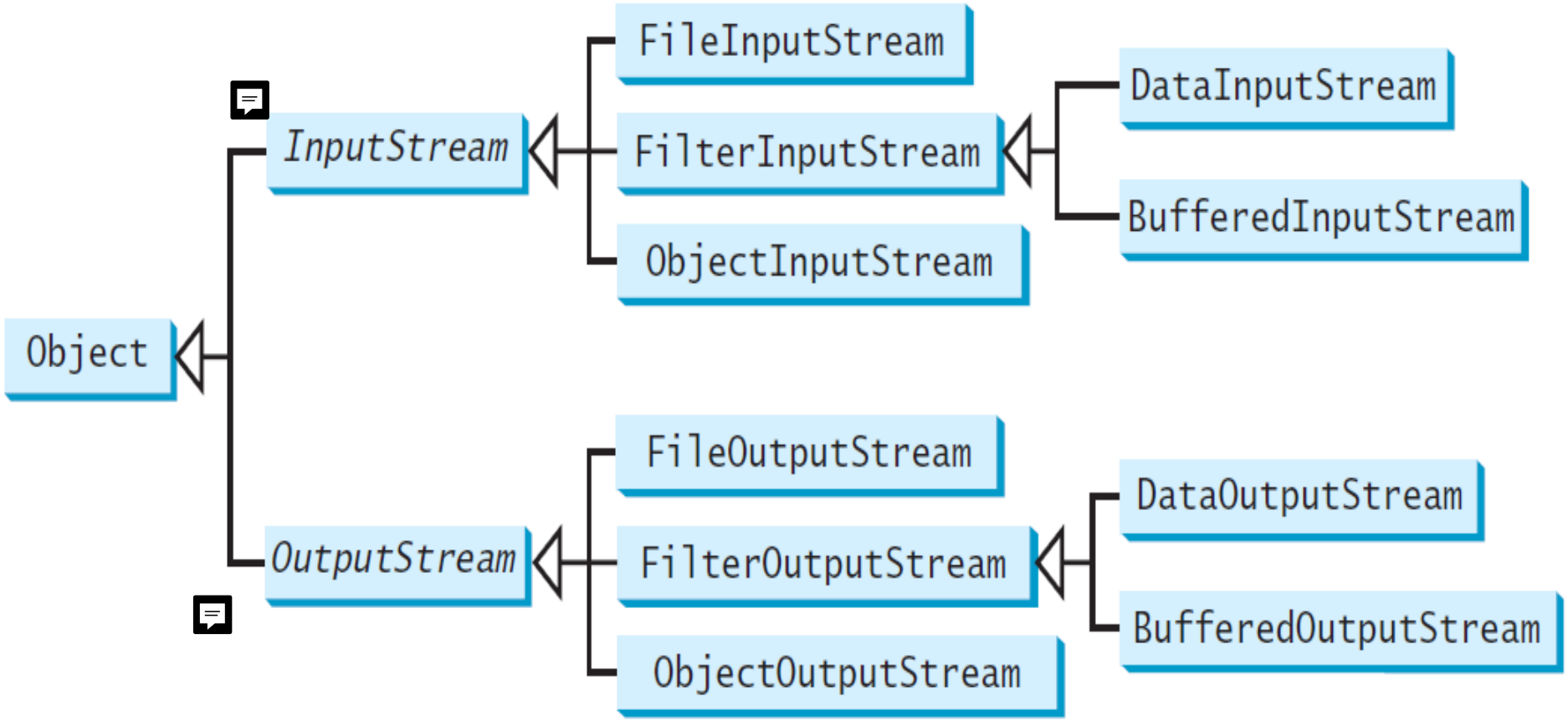


Binary I/O

- 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.



Binary I/O Classes



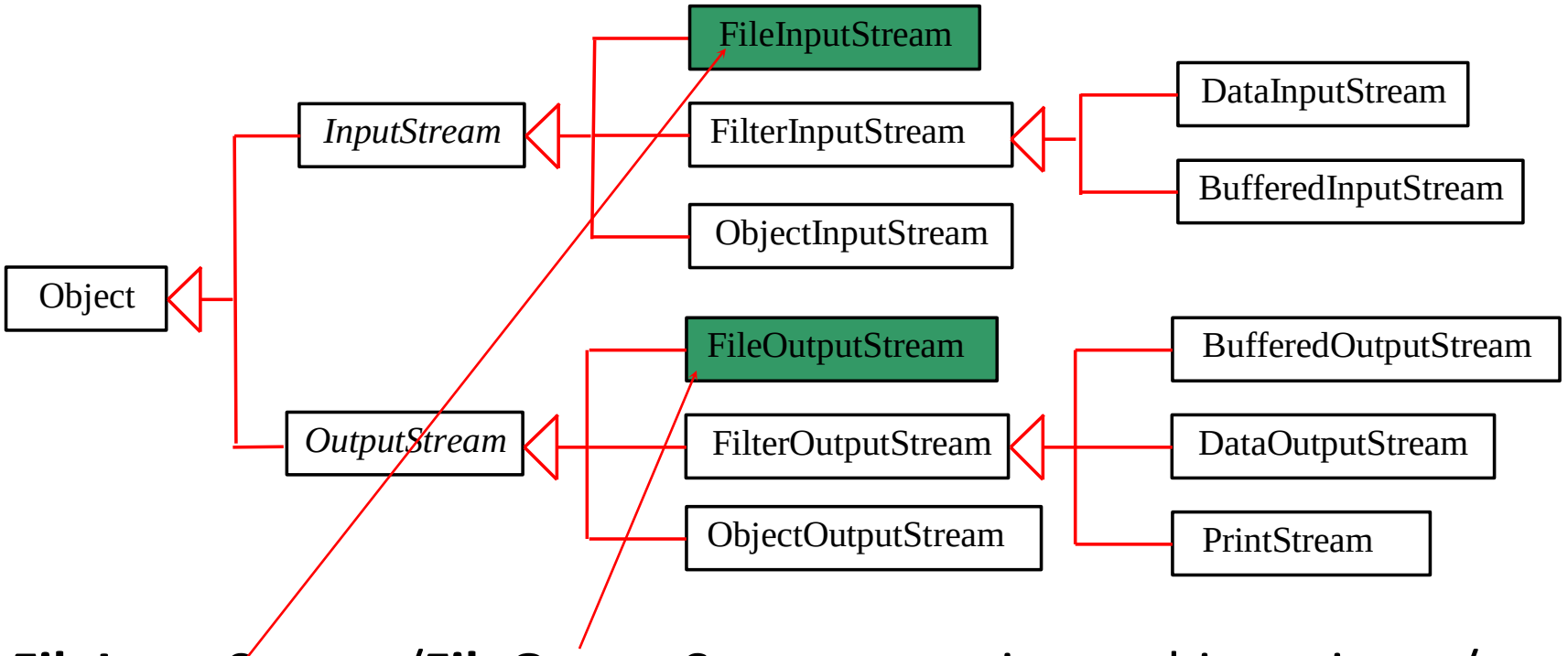
InputStream

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 0–255. 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 them in 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.
+close(): void	Closes this input stream and releases any system resources occupied by it.
+skip(n: long): long	Skips over and discards n bytes of data from this input stream. The actual number of bytes skipped is returned.

OutputStream

java.io.OutputStream	
+write(int b): void	Writes the specified byte to this output stream. The parameter `b` is an `int` value. `(byte)b` is written to the output stream.
+write(b: byte[]): void	Writes all the bytes in array `b` to the output stream.
+write(b: byte[], off: int, len: int): void	Writes `b[off]`, `b[off+1]`, ..., `b[off+len-1]` into the output stream.
+close(): void	Closes this output stream and releases any system resources occupied by it.
+flush(): void	Flushes this output stream and forces any buffered output bytes to be written out.

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.

FileInputStream

- 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

- To construct a **FileOutputStream**, use the following constructors:

  public FileOutputStream(String filename)

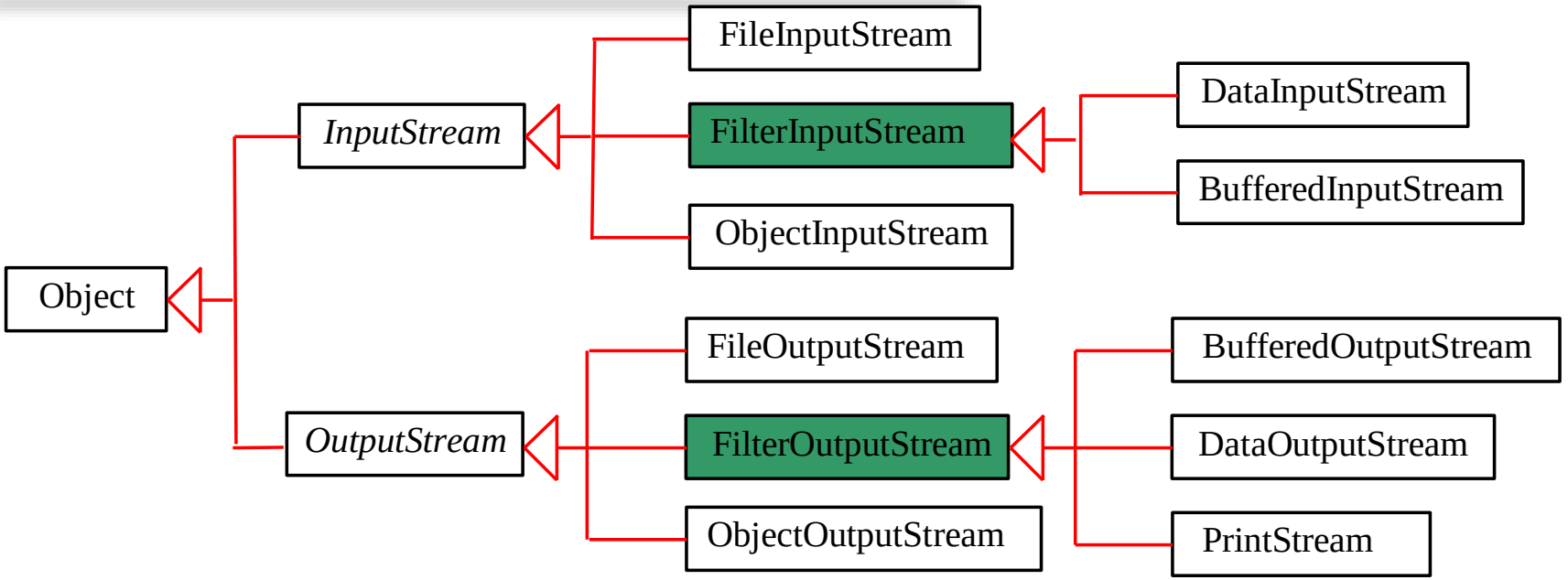
  public FileOutputStream(File file)

  public FileOutputStream(String filename, boolean append)

  public FileOutputStream(File file, boolean append)

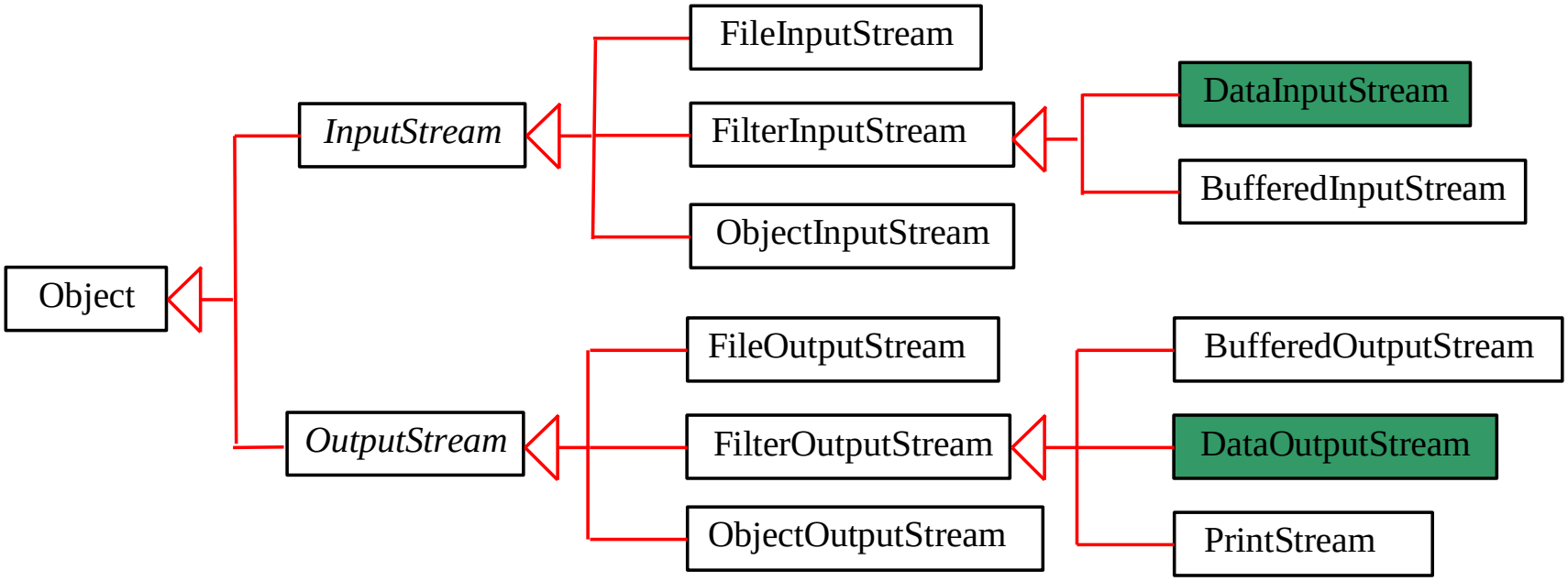
- If the file does not exist, a new file would be created.
- If the file already exists, the first two constructors would delete the current contents in the file.
- To keep the current content and append new data into the file, use the last two constructors by passing true to the append parameter.

FilterInputStream/FilterOutputStream



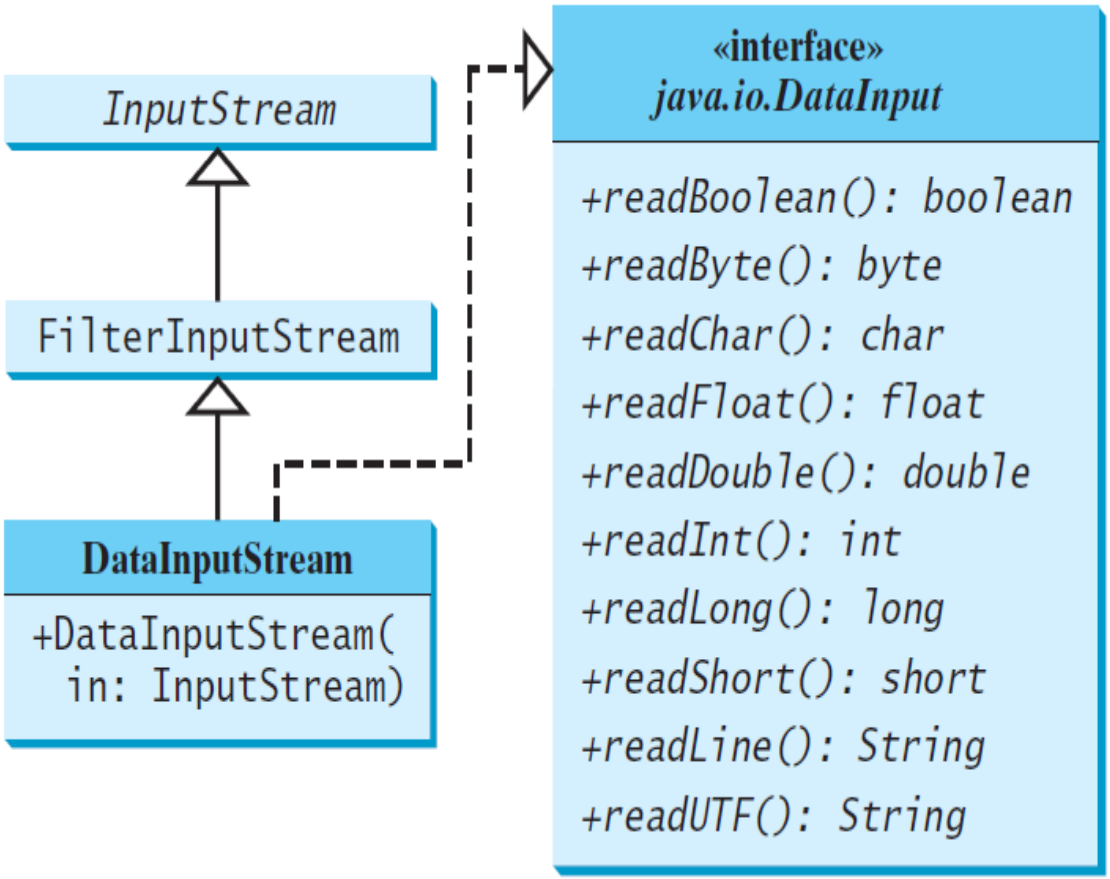
- Using a filter class enables you to read integers, doubles, and strings instead of bytes and characters.
- **FilterInputStream** and **FilterOutputStream** are the **base classes** for filtering data.

DataInputStream/DataOutputStream



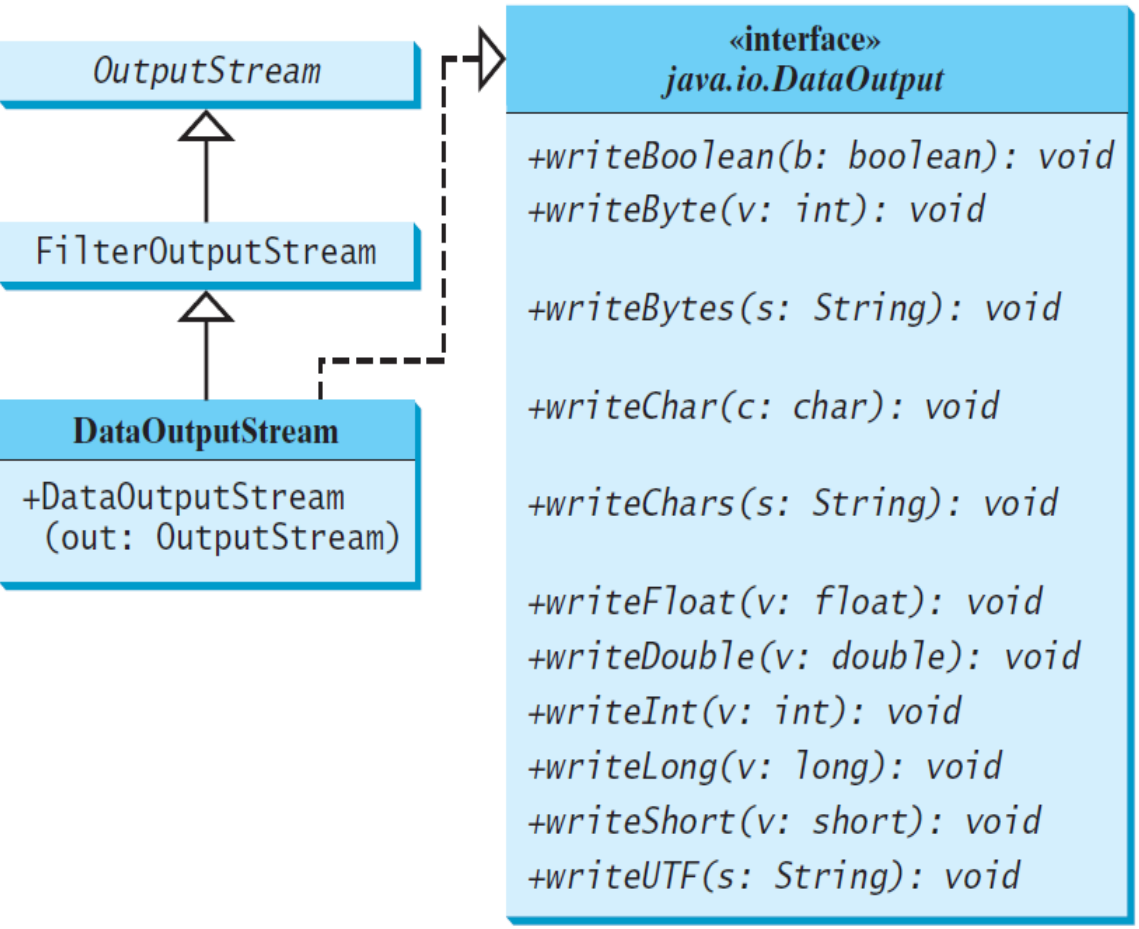
- **DataInputStream** reads bytes from the stream and converts them into appropriate primitive type values or strings.
- **DataOutputStream** converts primitive type values or strings into bytes and output the bytes to the stream.

DataInputStream



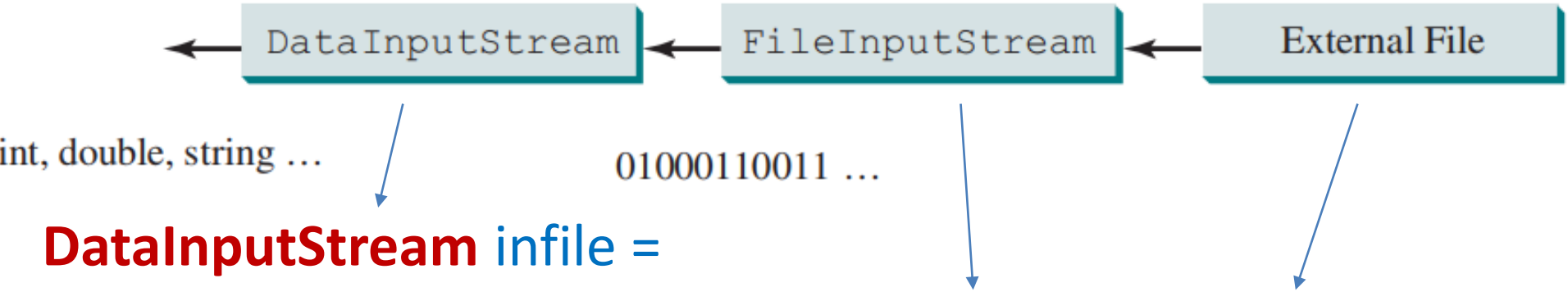
Reads a `Boolean` from the input stream.
Reads a `byte` from the input stream.
Reads a character from the input stream.
Reads a `float` from the input stream.
Reads a `double` from the input stream.
Reads an `int` from the input stream.
Reads a `long` from the input stream.
Reads a `short` from the input stream.
Reads a `line` of characters from input.
Reads a `string` in UTF format.

DataOutputStream

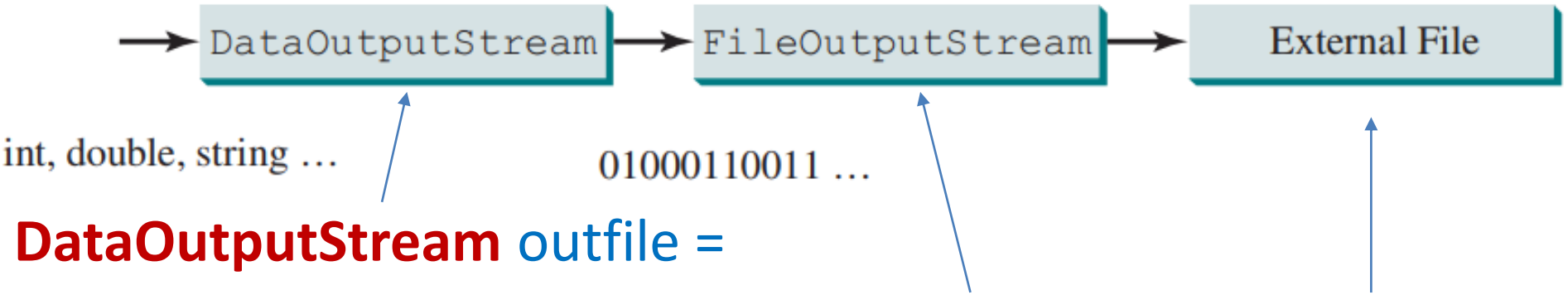


Writes a Boolean to the output stream.
Writes the eight low-order bits of the argument `v` to the output stream.
Writes the lower byte of the characters in a string to the output stream.
Writes a character (composed of 2 bytes) to the output stream.
Writes every character in the string `s` to the output stream, in order, 2 bytes per character.
Writes a `float` value to the output stream.
Writes a `double` value to the output stream.
Writes an `int` value to the output stream.
Writes a `long` value to the output stream.
Writes a `short` value to the output stream.
Writes `s` string in UTF format.

Using DataInputStream/DataOutputStream




```
DataInputStream infile =
  new DataInputStream(new FileInputStream("in.dat"));
```




```
DataOutputStream outfile =
  new DataOutputStream(new FileOutputStream("out.dat"));
```

Order and Format

- **CAUTION**: You have to read the data in the same order and same format in which they are stored.
- For example, since names are written in UTF-8 using **writeUTF**, you must read names using **readUTF**.

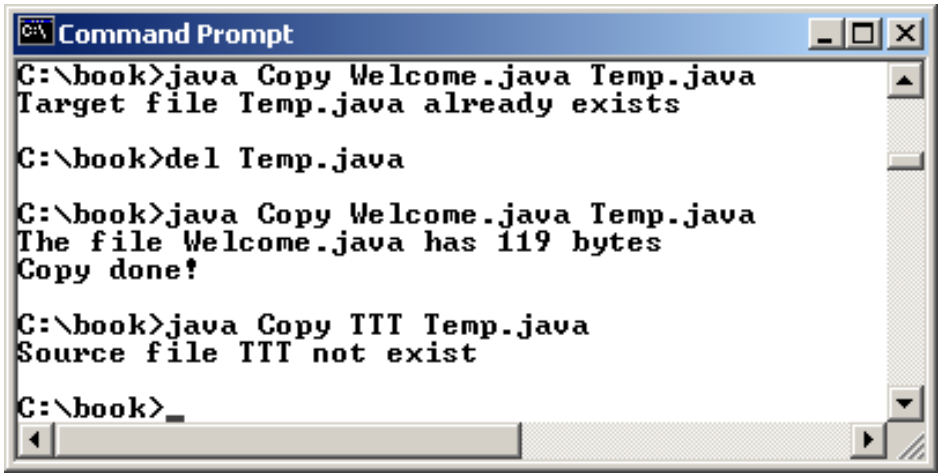
Checking End of File

- If you keep reading data at the end of a stream, an **EOFException** would occur. So how do you check the end of a file?
- You can use **input.available()** to check it.
- **input.available() == 0** indicates that it is the end of a file.

Case Studies: Copy File

This case study develops a program that copies files. The user needs to provide a source file and a target file as command-line arguments using the following command:

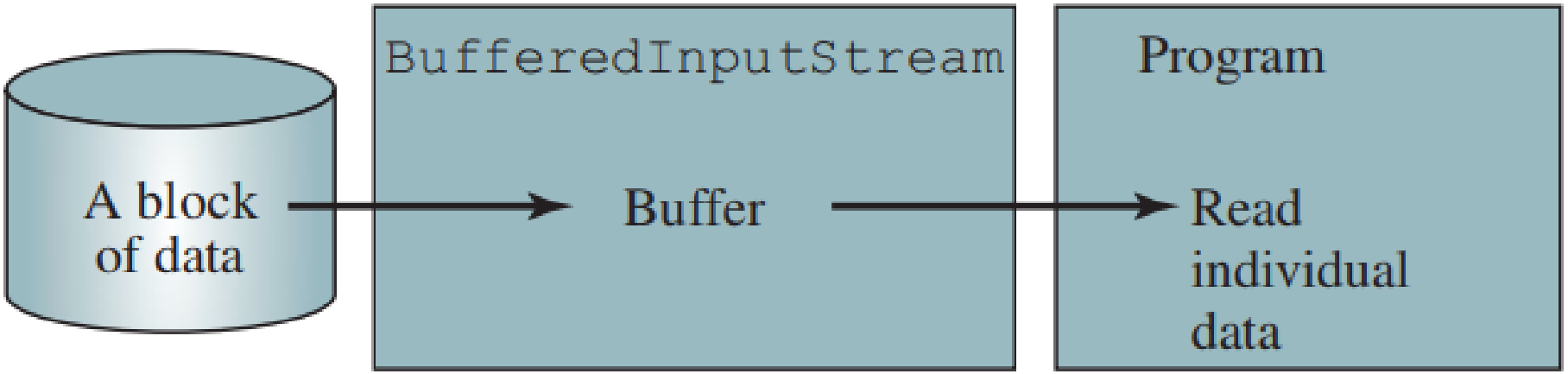
java Copy source target



The program copies a source file to a target file and displays the number of bytes in the file. If the source does not exist, tell the user the file is not found. If the target file already exists, tell the user the file already exists.

Self-Study

- **BufferedInputStream/BufferedOutputStream** can be used to speed up input and output by reducing the number of disk reads and writes.



Self-Study

- **ObjectInputStream/ObjectOutputStream** enables you to perform I/O for objects in addition to primitive-type values and strings.

```
try (FileOutputStream f = new
FileOutputStream("data.dat");
ObjectOutputStream output = new
ObjectOutputStream(f);) {
        output.writeUTF("Mamoun");
        output.writeDouble(55.5);
        output.writeObject(new Date());
}
```