Text & Binary IO

The **File** Class

- Data stored in the program are temporary; they are lost when the program terminates.

- To permanently store the data created in a program, you need to save them in a file on a disk or other permanent storage device.

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

The **File** Class (Cont.)

- An *absolute file name* (or *full name*) contains a file name with its complete path and drive letter.

- Absolute file names are machine dependent.

UNIX: **/home/seneca/FCET/JAC444S1Q/HelloWorld.java**

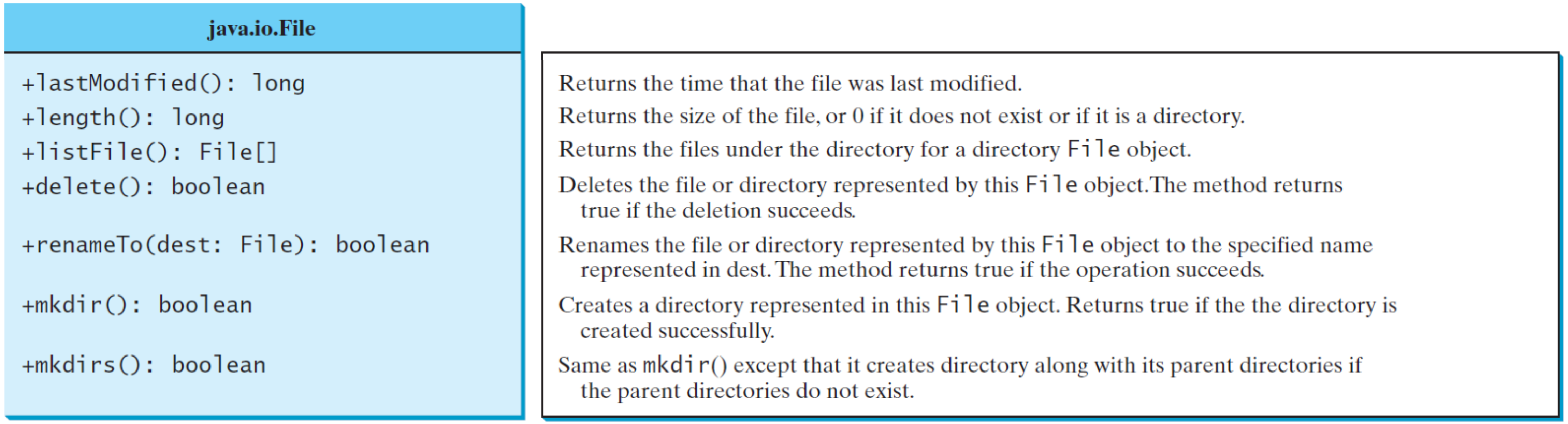
Windows: **c:\seneca\FCET\JAC444S1Q\HelloWorld.java**

- A *relative file name* is in relation to the current working directory.

The **File** Class (Cont.)

- The **File** class contains the methods for obtaining file and directory properties and for renaming and deleting files and directories.
- The **File** class does not contain the methods for reading and writing file contents.
- The file name is a string. The **File** class is a wrapper class for the file name and its directory path.

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.
`+getAbsolutePath(): String`	Returns the complete absolute file or directory name represented by the `File` object.
`+getCanonicalPath(): String`	Returns the same as `getAbsolutePath()` except that it removes redundant names, such as "." and "..", from the path name, resolves symbolic links (on Unix), and converts drive letters to standard uppercase (on Windows).
`+getName(): String`	Returns the last name of the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getName()` returns `test.dat`.
`+getPath(): String`	Returns the complete directory and file name represented by the `File` object. For example, new `File("c:\\book\\test.dat").getPath()` returns `c:\book\test.dat`.
`+getParent(): String`	Returns the complete parent directory of the current directory or the file represented by the `File` object. For example, new `File("c:\\book\\test.dat").getParent()` returns `c:\book`.



java.io.File	
`+lastModified(): long`	Returns the time that the file was last modified.
`+length(): long`	Returns the size of the file, or 0 if it does not exist or if it is a directory.
`+listFile(): File[]`	Returns the files under the directory for a directory `File` object.
`+delete(): boolean`	Deletes the file or directory represented by this `File` object.The method returns true if the deletion succeeds.
`+renameTo(dest: File): boolean`	Renames the file or directory represented by this `File` object to the specified name represented in dest. The method returns true if the operation succeeds.
`+mkdir(): boolean`	Creates a directory represented in this `File` object. Returns true if the the directory is created successfully.
`+mkdirs(): boolean`	Same as `mkdir`() except that it creates directory along with its parent directories if the parent directories do not exist.

The **File** Class (Cont.)

- **new File("c:\\book")** creates a **File** object for the directory **c:\book**, and **new File("c:\\book\\test.dat")** creates a **File** object for the file **c:\book\test.dat**, both on Windows.

- You can use the **File** class's **isDirectory()** method to check whether the object represents a directory, and the **isFile()** method to check whether the object represents a file.

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

The **File** Class – Example

- This example, `TestFileClass.java,` demonstrates how to create a **File** object and use the methods in the **File** class to obtain its properties.

- The program creates a **File** object for the file **senecaatyorkcampus.jpg**. This file is stored under the **images** directory in the current directory.

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.

- A **File** object encapsulates the properties of a file or a path, but it does not contain the methods for creating a file or for writing/reading data to/from a file (referred to as data *input* and *output*, or I/O for short).

- In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file.

File Input and Output (Cont.)

- There are two types of files: text and binary.

- Text files are essentially characters on disk.

- Here, we will introduce how to read/write strings and numeric values from/to a text file using the **Scanner** and **PrintWriter** classes.

Writing Data Using **PrintWriter**

- The **java.io.PrintWriter** class can be used to create a file and write data to a text file.

- First, you have to create a **PrintWriter** object for a text file as follows:

  ```
  PrintWriter output = new PrintWriter(filename);
  ```

- Then, you can invoke the **print**, **println**, and **printf** methods on the **PrintWriter** object to write data to a file.

Writing Data Using **PrintWriter** – Example

- This example, `WriteData.java` , creates an instance of **PrintWriter** and writes two lines to the file **scores.txt**.

- Each line consists of a first name (a string), a middle-name initial (a character), a last name (a string), and a score (an integer).

Closing Resources Automatically Using 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.

```
try (declare and create resources) {
    Use the resource to process the file;
}
```

- Using the try-with-resources syntax, we rewrite the code in `WriteData.java` in `WriteDataWithAutoClose.java`.

Closing Resources Automatically Using try-with-resources (Cont.)

- A resource is declared and created followed by the keyword **try**.
- Note that the resources are enclosed in the parentheses.
- The resources must be a subtype of **AutoCloseable** such as a **PrinterWriter** that has the **close()** method.
- A resource must be declared and created in the same statement and multiple resources can be declared and created inside the parentheses.
- The statements in the block immediately following the resource declaration use the resource.
- After the block is finished, the resource’s **close()** method is automatically invoked to close the resource.

Reading Data Using **Scanner**

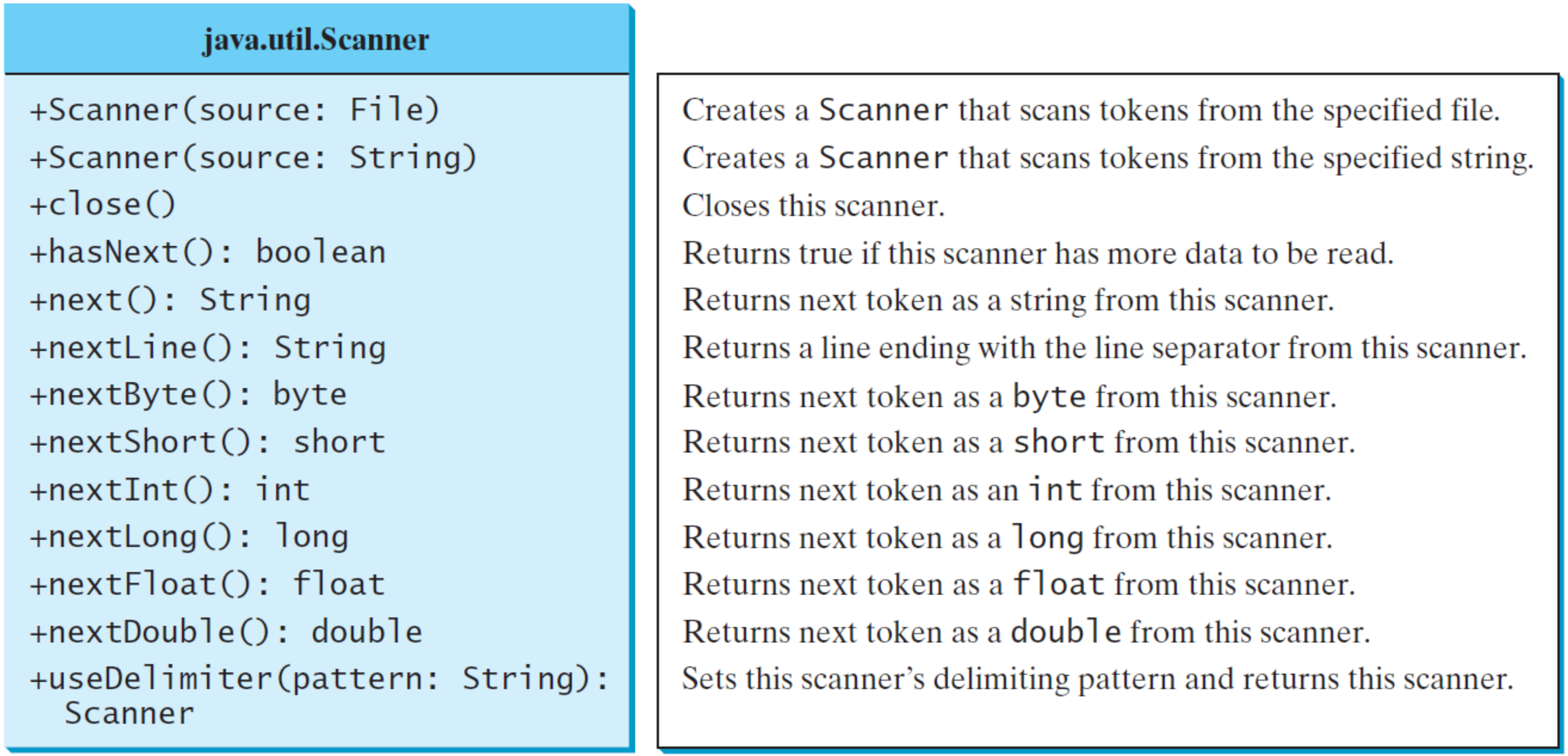
- The java.util.Scanner class was used to read strings and primitive values from the console.

- A Scanner breaks its input into tokens delimited by whitespace characters. To read from the keyboard, you create a Scanner for System.in, as follows:

```
Scanner input = new Scanner(System.in);
```

- To read from a file, create a **Scanner** for a file, as follows:

```
Scanner input = new Scanner(new File(filename));
```



java.util.Scanner	
`+Scanner(source: File)`	Creates a `Scanner` that scans tokens from the specified file.
`+Scanner(source: String)`	Creates a `Scanner` that scans tokens from the specified string.
`+close()`	Closes this scanner.
`+hasNext(): boolean`	Returns true if this scanner has more data to be read.
`+next(): String`	Returns next token as a string from this scanner.
`+nextLine(): String`	Returns a line ending with the line separator from this scanner.
`+nextByte(): byte`	Returns next token as a `byte` from this scanner.
`+nextShort(): short`	Returns next token as a `short` from this scanner.
`+nextInt(): int`	Returns next token as an `int` from this scanner.
`+nextLong(): long`	Returns next token as a `long` from this scanner.
`+nextFloat(): float`	Returns next token as a `float` from this scanner.
`+nextDouble(): double`	Returns next token as a `double` from this scanner.
`+useDelimiter(pattern: String): Scanner`	Sets this scanner's delimiting pattern and returns this scanner.

Reading Data Using **Scanner** – Example

- This example, `ReadData.java`, creates an instance of **Scanner** and reads data from the file **scores.txt**.

Example: TestScannerMethods.java

- Test Scanner methods.

Case Study: Replacing Text

- Suppose you are to write an application named **ReplaceText.java** that replaces all occurrences of a string in a text file with a new string. The file name and strings are passed as command-line arguments as follows:

```
java ReplaceText sourceFile targetFile oldString newString
```

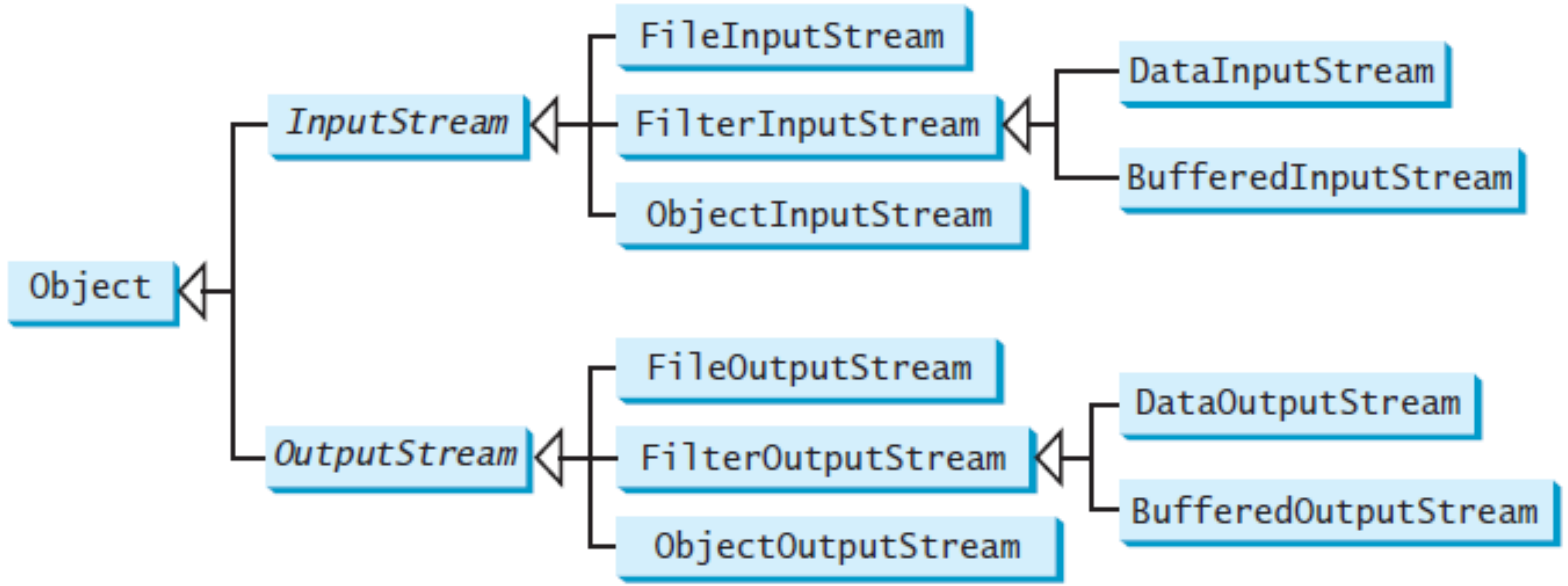
- For example, invoking

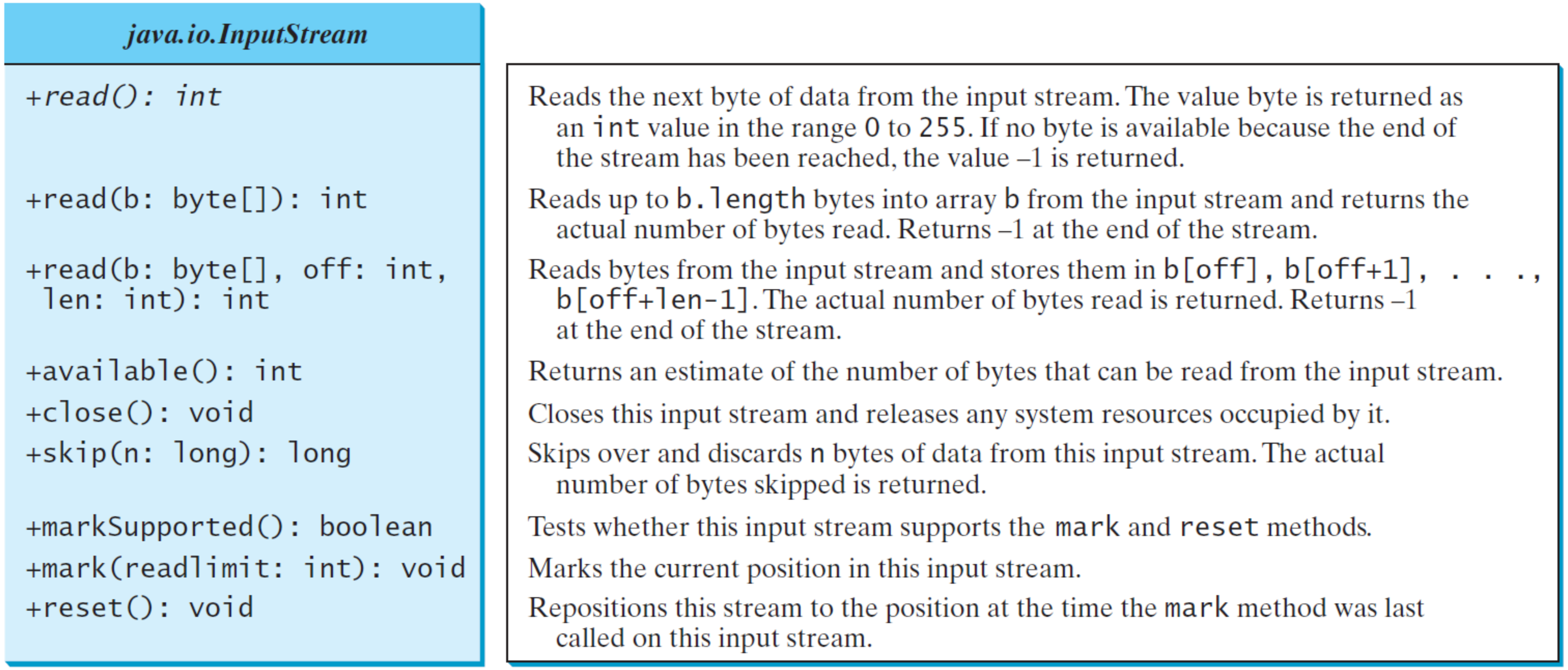
```
java ReplaceText source.txt destination.txt Hossein James
```

- replaces all the occurrences of **Hossein** by **James** in the file **source.txt** and saves the new file in **destination.txt**.

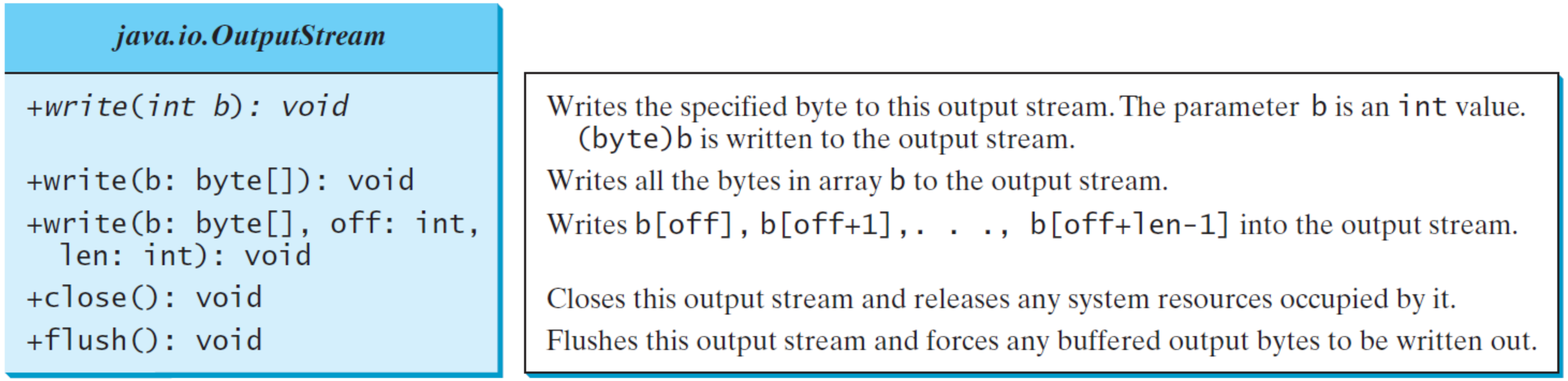
Binary I/O Classes

- The abstract **InputStream** is the root class for reading binary data, and the abstract **OutputStream** is the root class for writing binary data.





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 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.
`+available(): int`	Returns an estimate of the number of bytes that can be read from the input 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.
`+markSupported(): boolean`	Tests whether 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.



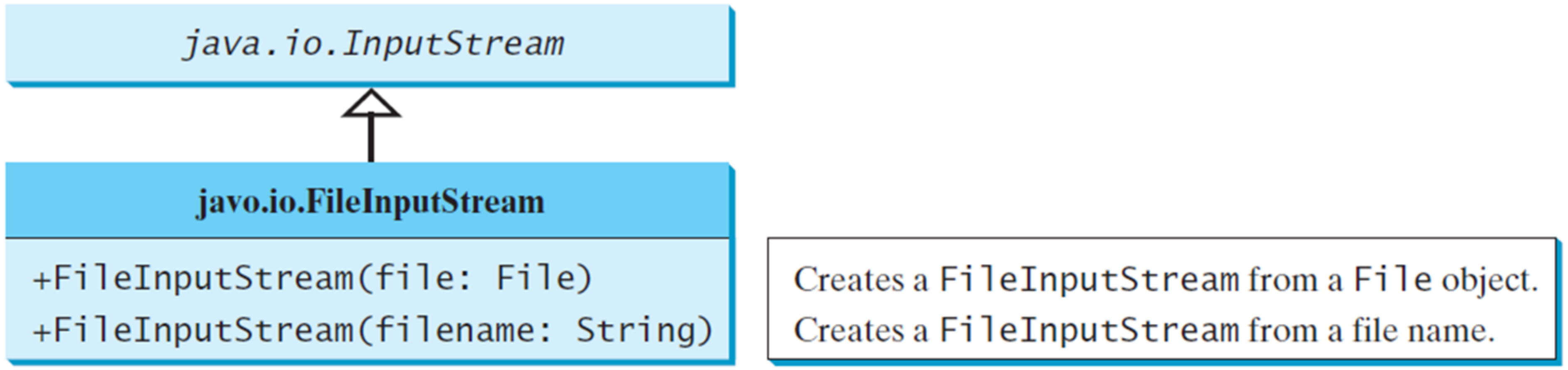
java.io.OutputStream	
`+write(int b): void`	Writes the specified byte to this output stream. The parameter `b` is an `int` value. `(byte)b` is written to the output stream.
`+write(b: byte[]): void`	Writes all the bytes in array `b` to the output stream.
`+write(b: byte[], off: int, len: int): void`	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

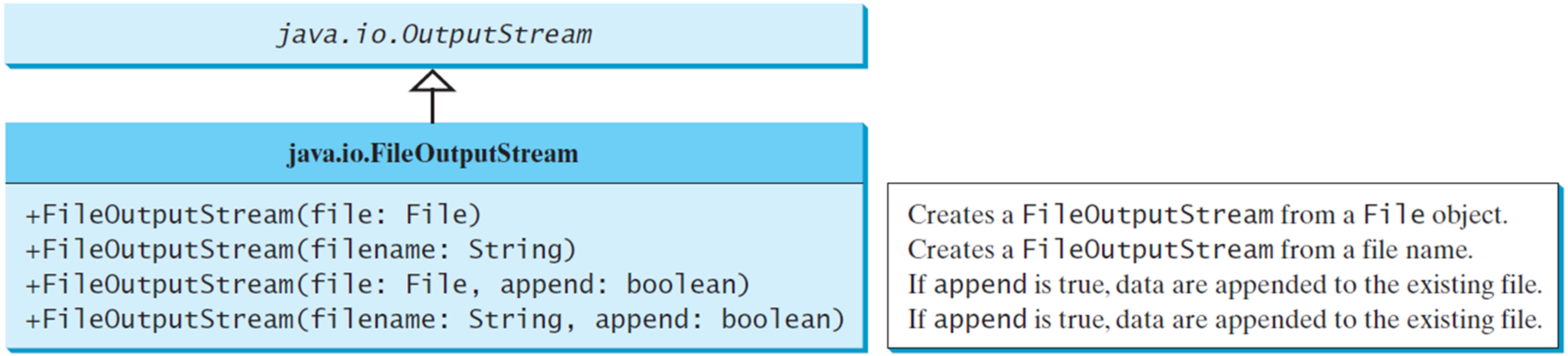
- **FileInputStream/FileOutputStream** is for reading/writing bytes from/to **files**.
- All the methods in these classes are inherited from **InputStream** and **OutputStream**.
- **FileInputStream/FileOutputStream** does not introduce new methods.
- To construct a **FileInputStream**, use its constructors.
- A **java.io.FileNotFoundException** will occur if you attempt to create a **FileInputStream** with a nonexistent file.

FileInputStream/FileOutputStream

- To construct a **FileOutputStream**, use its constructors.
- If the file does not exist, a new file will be created.
- If the file already exists, the first two constructors will delete the current content of the file.
- To retain the current content and append new data into the file, use the last two constructors and pass **true** to the **append** parameter.


java.io.InputStream
javo.io.FileInputStream
+FileInputStream(file: File)
+FileInputStream(filename: String)

Creates a `FileInputStream` from a `File` object.
Creates a `FileInputStream` from a file name.


java.io.OutputStream
java.io.FileOutputStream
+FileOutputStream(file: File)
+FileOutputStream(filename: String)
+FileOutputStream(file: File, append: boolean)
+FileOutputStream(filename: String, append: boolean)

Creates a `FileOutputStream` from a `File` object.
Creates a `FileOutputStream` from a file name.
If `append` is true, data are appended to the existing file.
If `append` is true, data are appended to the existing file.

FileInputStream/FileOutputStream

- Almost all the methods in the I/O classes throw **java.io.IOException**. Therefore, you have to declare to throw **java.io.IOException** in the method or place the code in a try-catch block, as shown below:

Declaring exception in the method

```
public static void main(String[] args)
    throws IOException {
  // Perform I/O operations
}
```

Using try-catch block

```
public static void main(String[] args) {
  try {
    // Perform I/O operations
  }
  catch (IOException ex) {
    ex.printStackTrace();
  }
}
```

Example: TestFileStream.java

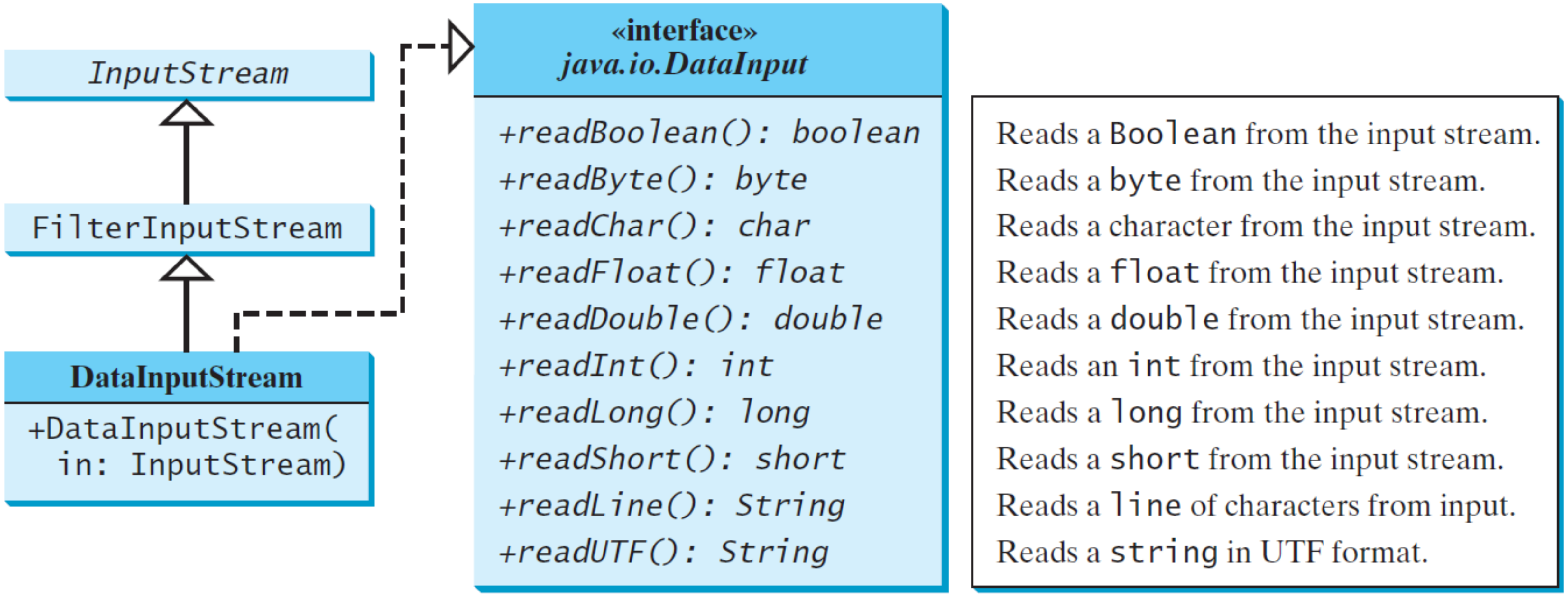
- It uses binary I/O to write ten byte values from **10** to **1** to a file named **temp.dat** and reads them back from the file.

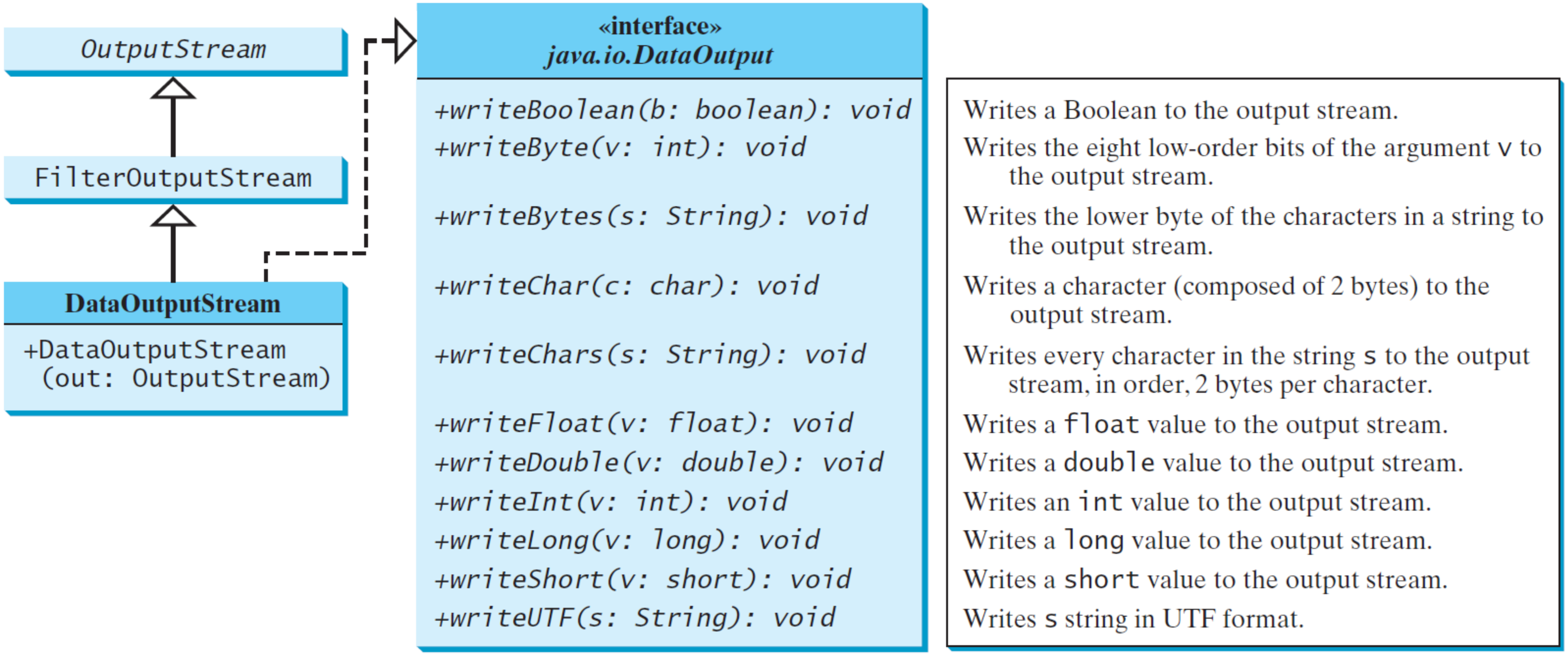
FilterInputStream/FilterOutputStream

- ***Filter streams*** are streams that filter bytes for some purpose.
- The basic byte input stream provides a **read** method that can be used only for reading bytes.
- If you want to read integers, doubles, or strings, you need a filter class to wrap the byte input stream.
- 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.
- When you need to process primitive numeric types, use **DataInputStream** and **DataOutputStream** to filter bytes.

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 outputs the bytes to the stream.
- **DataInputStream** extends **FilterInputStream** and implements the **DataInput** interface.
- **DataOutputStream** extends **FilterOutputStream** and implements the **DataOutput** interface.
- **DataInputStream** implements the methods defined in the **DataInput** interface to read primitive data-type values and strings.
- **DataOutputStream** implements the methods defined in the **DataOutput** interface to write primitive data-type values and strings.


InputStream
FilterInputStream
DataInputStream
+DataInputStream(
in: InputStream)
«interface»
java.io.DataInput
+readBoolean(): boolean
+readByte(): byte
+readChar(): char
+readFloat(): float
+readDouble(): double
+readInt(): int
+readLong(): long
+readShort(): short
+readLine(): String
+readUTF(): String
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.


OutputStream
FilterOutputStream
DataOutputStream
+DataOutputStream (out: OutputStream)
«interface»
java.io.DataOutput
+writeBoolean(b: boolean): void
+writeByte(v: int): void
+writeBytes(s: String): void
+writeChar(c: char): void
+writeChars(s: String): void
+writeFloat(v: float): void
+writeDouble(v: double): void
+writeInt(v: int): void
+writeLong(v: long): void
+writeShort(v: short): void
+writeUTF(s: String): void
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.

Example: TestDataStream.java

- It writes student names and scores to a file named **temp.dat** and reads the data back from the file.

Detecting the End of a File

- If you keep reading data at the end of an **InputStream**, an **EOFException** will occur.

- This exception can be used to detect the end of a file, as you will see in **DetectEndOfFile.java** example.