CS-2012 Introduction to Programming II

Binary I/O

California State University, Los Angeles
Computer Science

Introduction to Binary I/O

Introduction

- files can be classified as text or binary files
  - ***text files***:
    - any file that can be processed by a text editor
    - characters are encoded using ASCII or Unicode values
  - ***binary files***:
    - all other files, these can't be read by a text editor
    - more efficient to process than text files.

- Examples:
  - .java files (the source code files) are stored in text files.
  - .class files (the compiled code files) are stored in binary files.

How Java Handles Text I/o

Text I/O

- Remember: Java has a **`File`** class which encapsulates the properties of a file or path.
- **`File`** does not have methods to read/write from/to files.
- If you want file I/O you need other Java classes.

Text I/O

- Example:

```
PrintWriter output = new PrintWriter("temp.txt")
```

- Can now use the **`print()`** method of the **`PrintWriter`** object to write to the file.

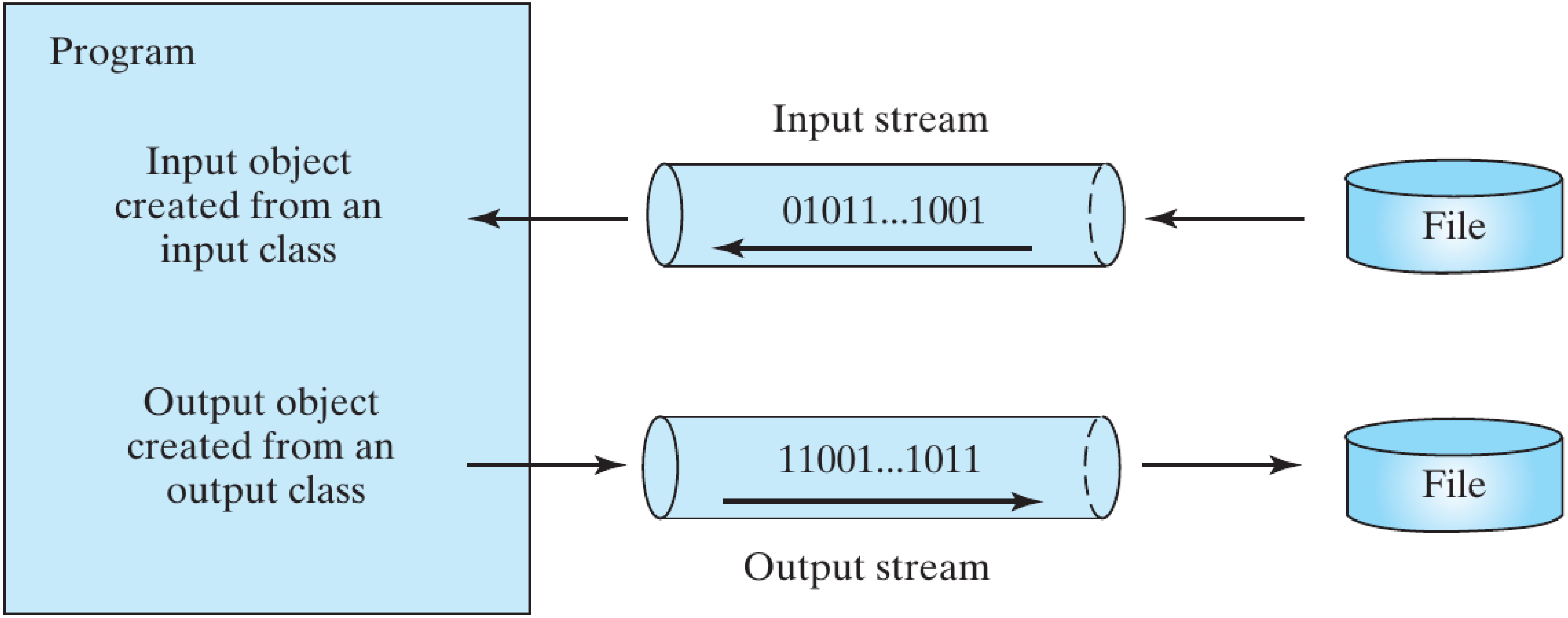
```
output.print("Java 101");
//writes "Java 101" to the file.
```

- **`Scanner`** can also be used to read/write text files.

- These classes work with ***streams*** of data.

- An input object is also called an ***input stream*** and an output object is an ***output stream***

Text I/O


Program
Input object created from an input class
Output object created from an output class
Input stream
01011...1001
File
11001...1011
File
Output stream

The program receives data through an input object and sends data through an output object.

Text I/O vs. Binary I/O

Text IO / vs. Binary I/O

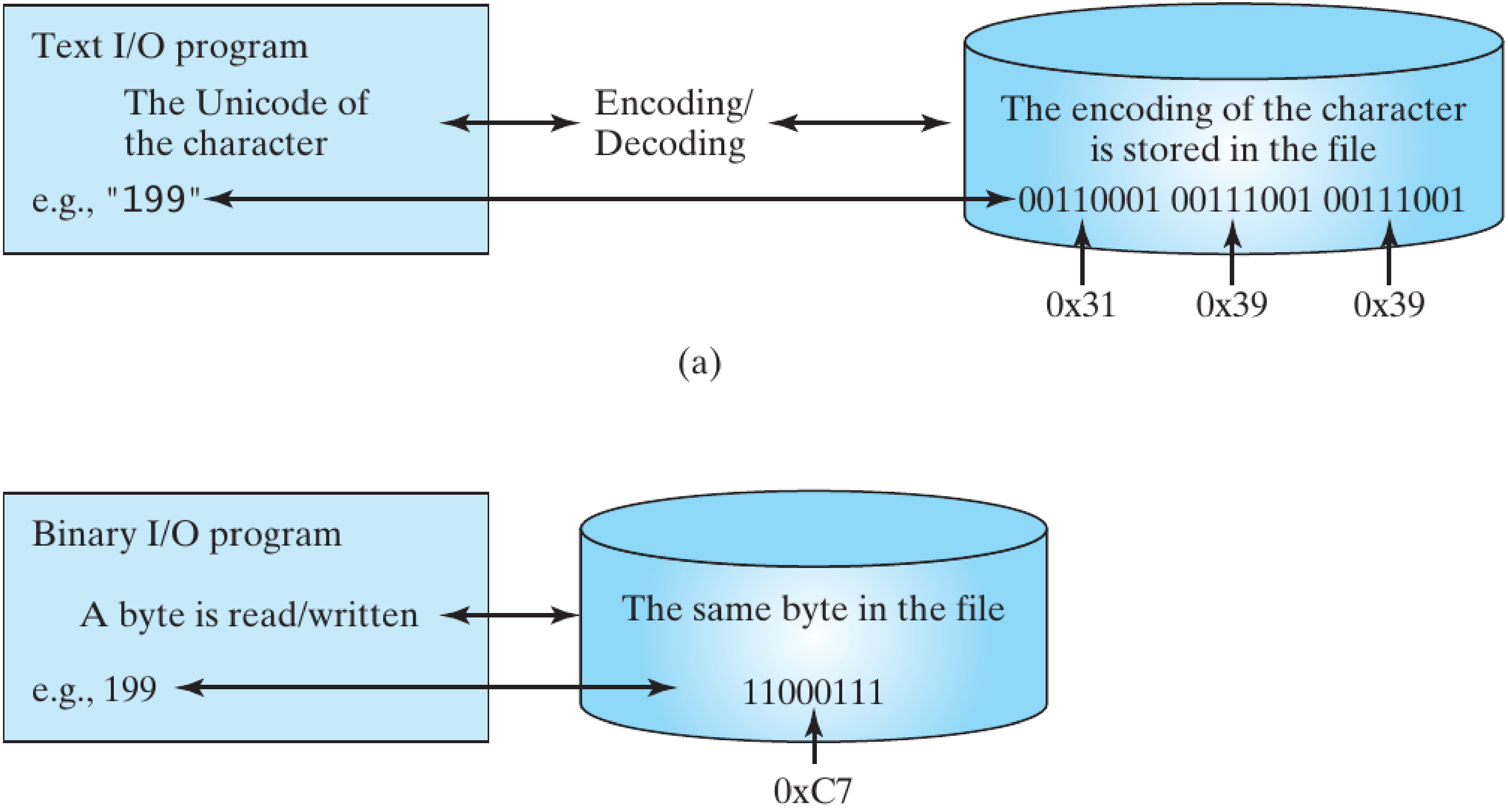
- Computers do not differentiate between text and binary files.
  - all files are stored in a binary format

- Text I/O is built on top of binary I/O to provide a level of abstraction for character encoding and decoding

- Encoding and decoding are performed automatically
  - JVM converts Unicode to a file-specific encoding when writing
  - JVM converts a file-specific encoding to Unicode when reading

- Binary I/O does not require conversions
  - writing a numeric value for example writes the exact value stored in memory
  - example: 199 could be represented as 0xC7 in memory and 0xC7 is what gets written to the file.
  - this is why Binary I/O is more efficient.

Text vs. Binary I/O



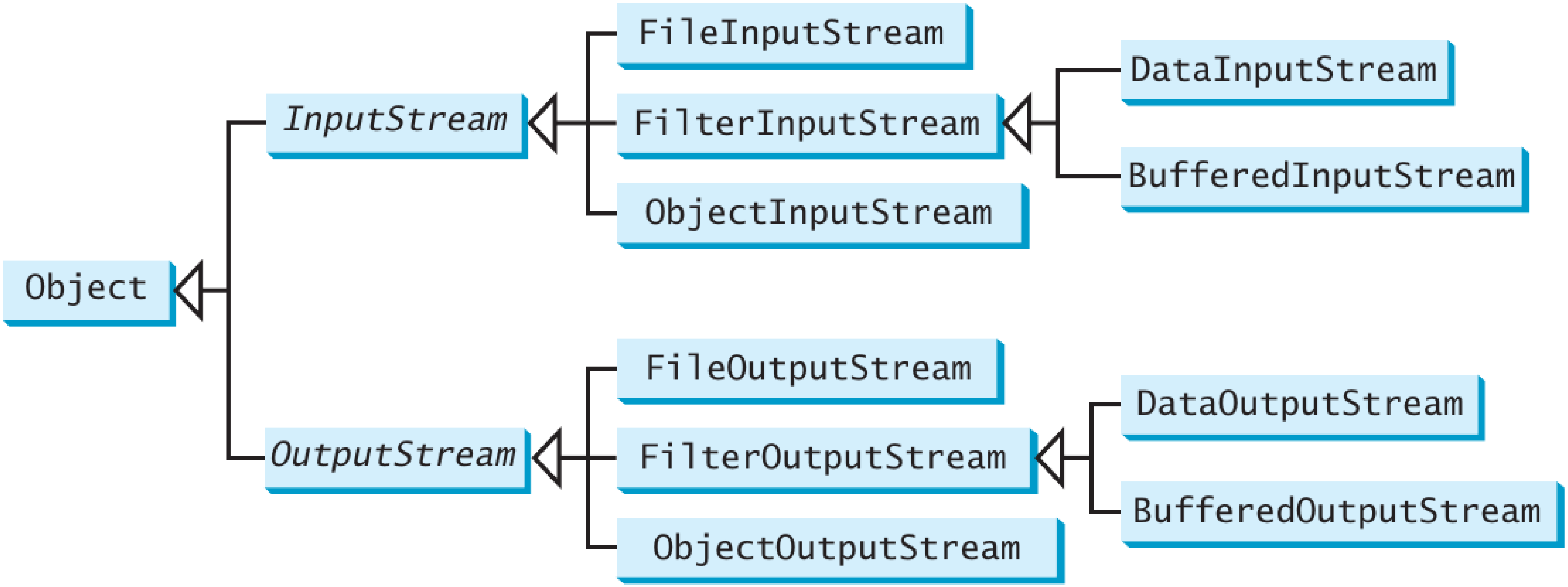
(b)

Text I/O requires encoding and decoding, whereas binary I/O does not.

Binary I/O Classes

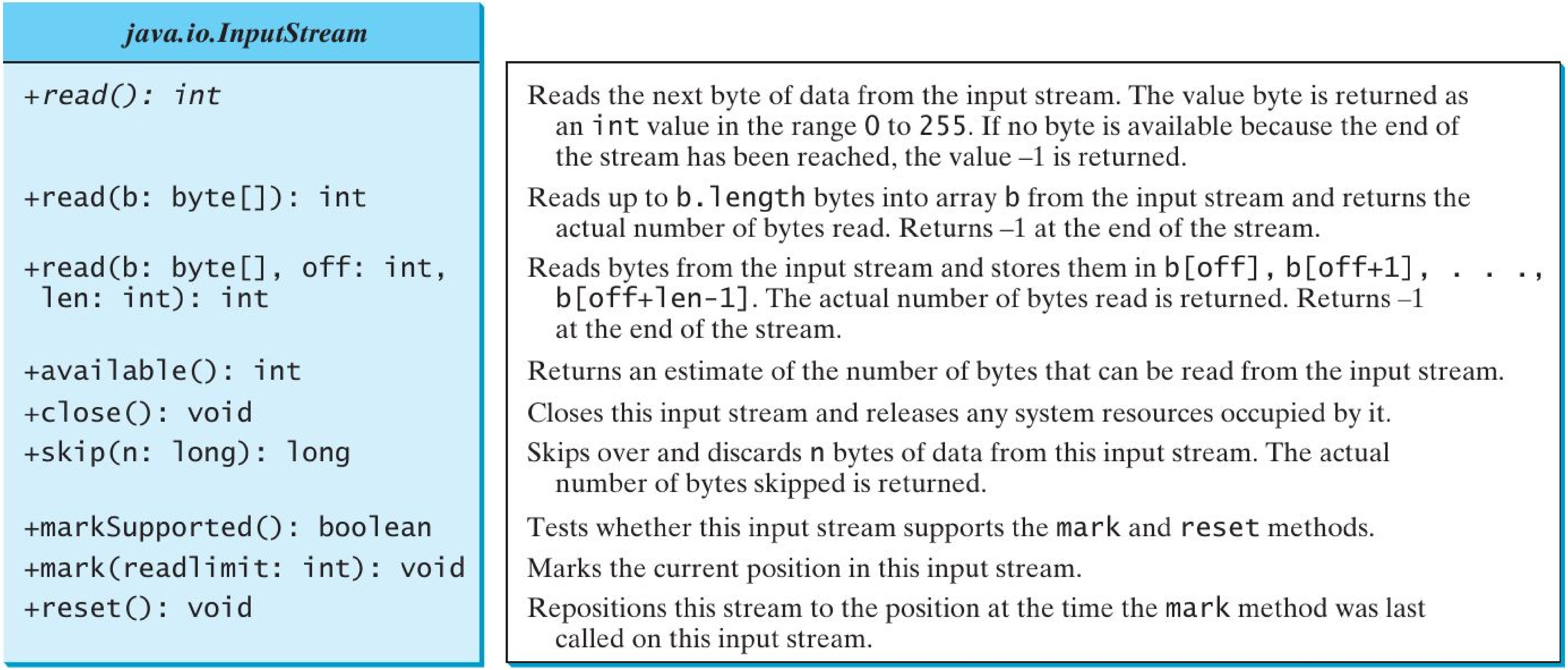
Binary I/O Classes

- The I/O Classes are great examples of applying inheritance (superclasses and subclasses).
- **`InputStream`** is the root for the binary input classes.
- **`OutputStream`** is the root for the binary output classes.



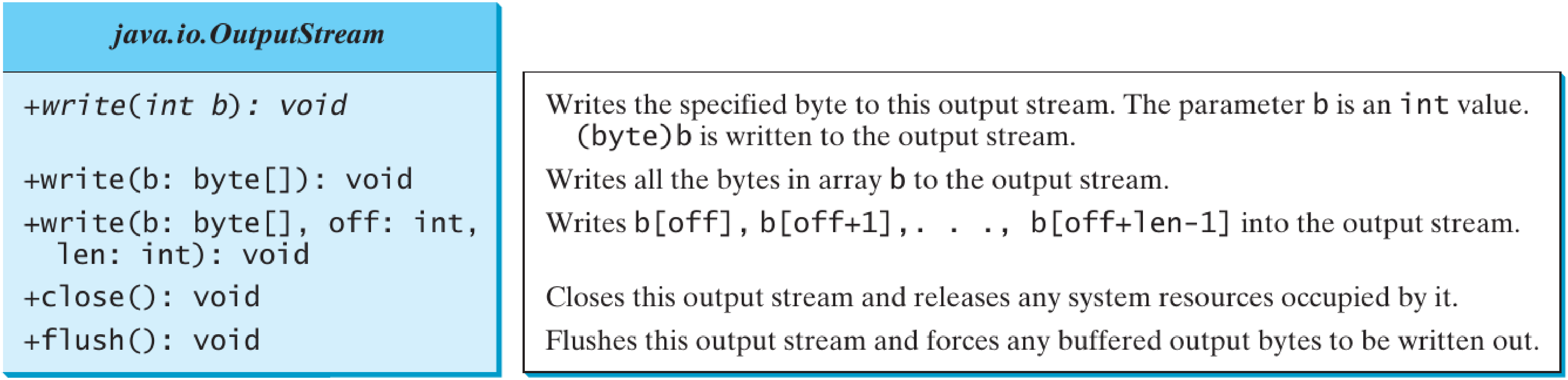
`InputStream`, **`OutputStream`**, and their subclasses are for performing binary I/O.

InputStream Methods



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.

`OutputStream` Methods



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.

- NOTE: All methods in the binary I/O classes throw **`java.io.IOException`** or a subclass of **`IOException`**

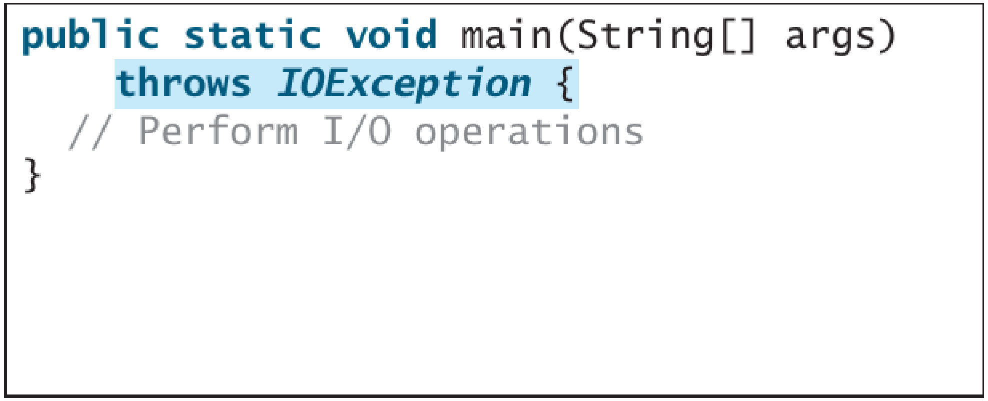
FileInputStream / FileOutputStream

- **`FileInputStream`** / **`FileOutputStream`** are for reading/writing bytes from / to files.

- All methods inherited from **`InputStream`** and **`OutputStream`**, no new methods are introduced.

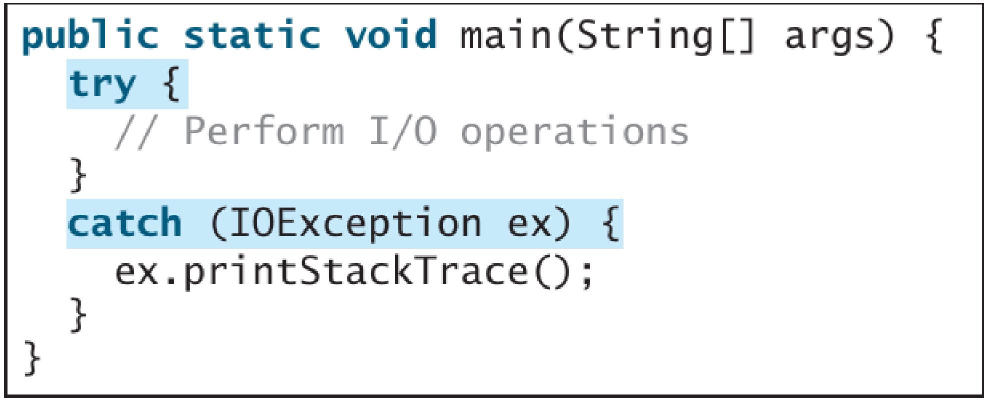
- Don't forgot to use exception handling when working with these classes.

Declaring exception in the method



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

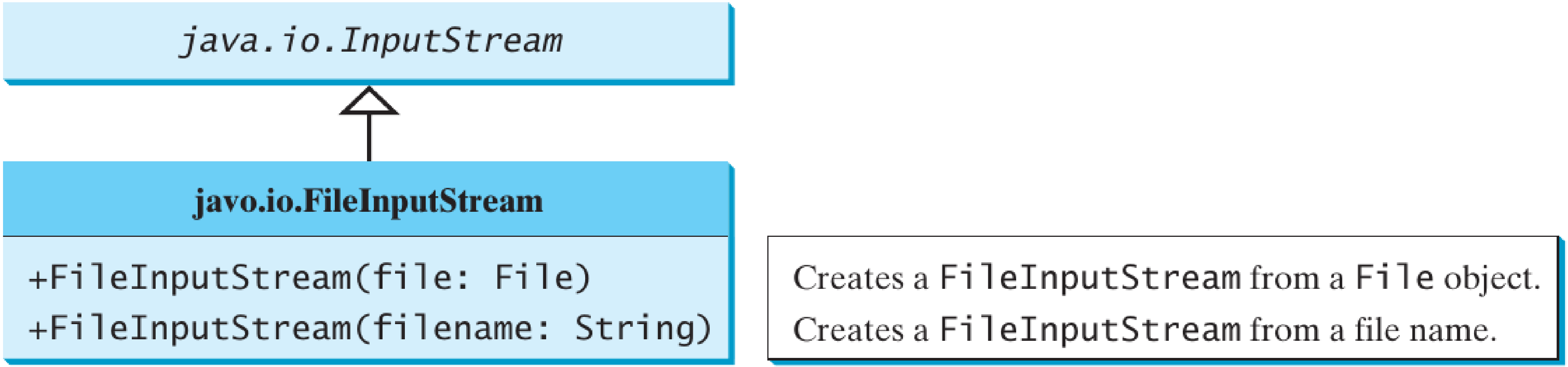
Using try-catch block



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

`FileInputStream`

- Use the following Constructors to make a **`FileInputStream`** object



java.io.InputStream

javo.io.FileInputStream

+FileInputStream(file: File)

+FileInputStream(filename: String)

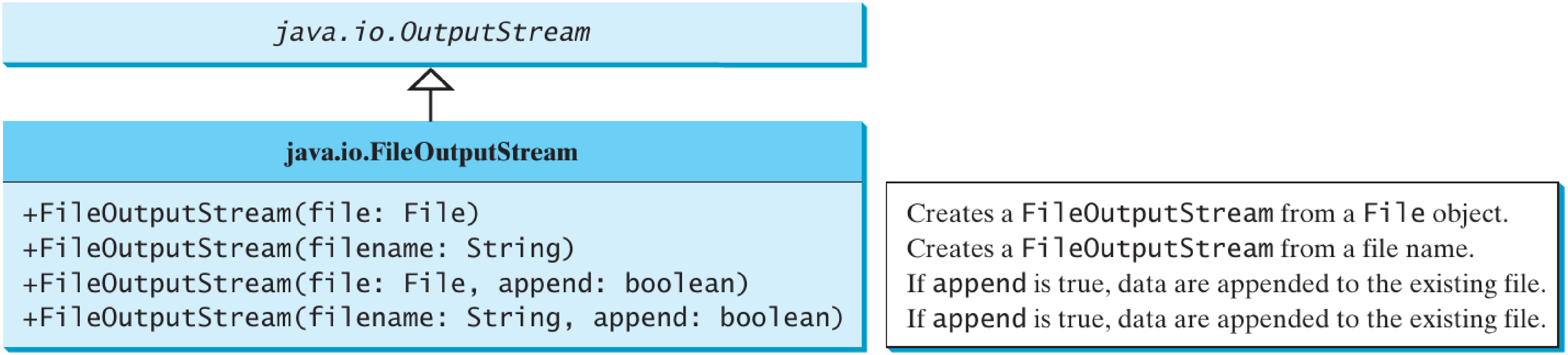
Creates a FileInputStream from a File object.

Creates a FileInputStream from a file name.

- If you make an instance of **`FileInputStream`** and it can't find the file, a **`FileNotFoundException`** is thrown.

`FileOutputStream`

- Use the following Constructors to make a **`FileOutputStream`** object.



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.

- If you make an instance of **`FileOutputStream`** it will create a new file if the one given does not exist.
- If the file already exists the contents of the old file will be deleted.
  - to retain the contents of the file and add data tot the end of the file, pass **`true`** to the append parameter of the constructor.

`FileInputStream` / `FileOutputStream` Example

- See Code: **`TestFileStream.java`**

`FilterInputStream` / `FilterOutputStream`

- ***filter streams*** filter bytes for some specific purpose

- the basic byte input stream provides a **`read()`** method that can only read bytes

- if you want to read another data type you need a filter class to wrap the byte input stream.

- **`FilterInputStream`** and **`FilterOutputStream`** are the base classes for filtering data.

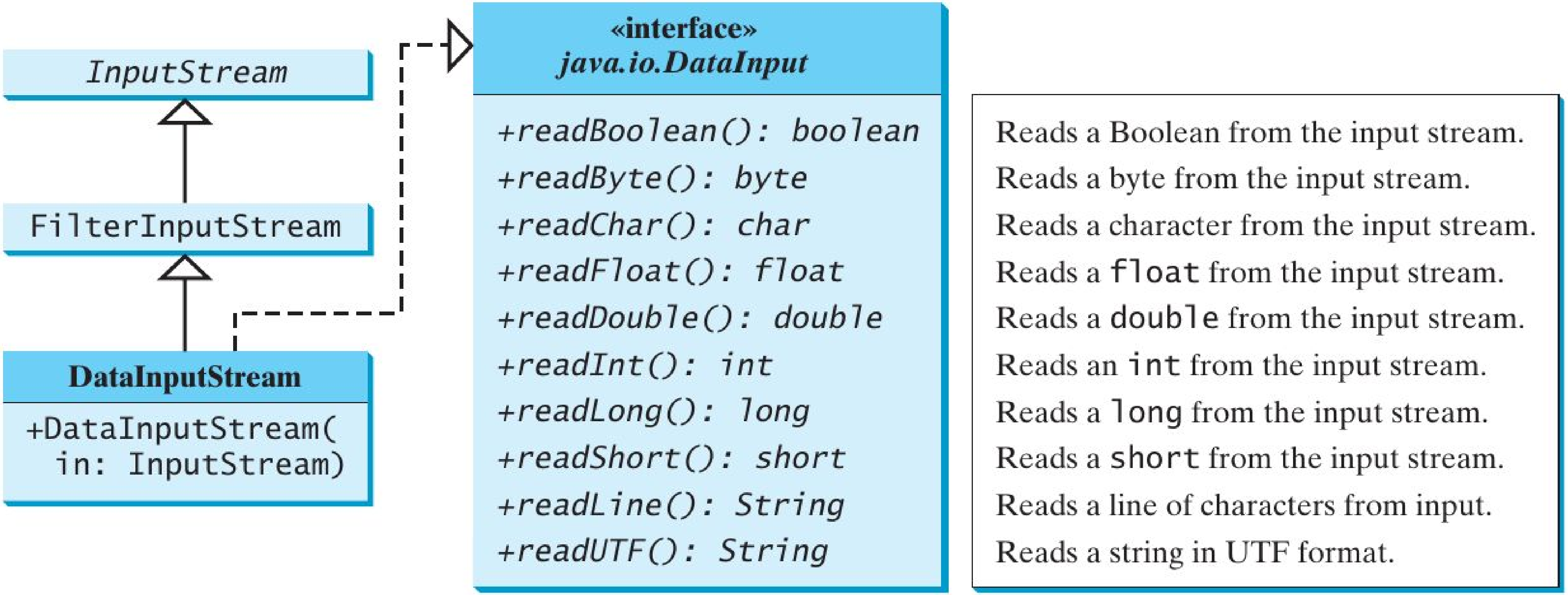
- Use **`DataInputStream`** and **`DataOutputStream`** to filter bytes to ints, doubles, or strings instead of bytes.

DataInputStream / DataOutputStream

- DataInputStream: reads bytes converts them to primitive values or strings

- DataOutputStream: converts primitive types or strings into bytes and writes them.

DataInputStream



InputStream

FilterInputStream

DataInputStream

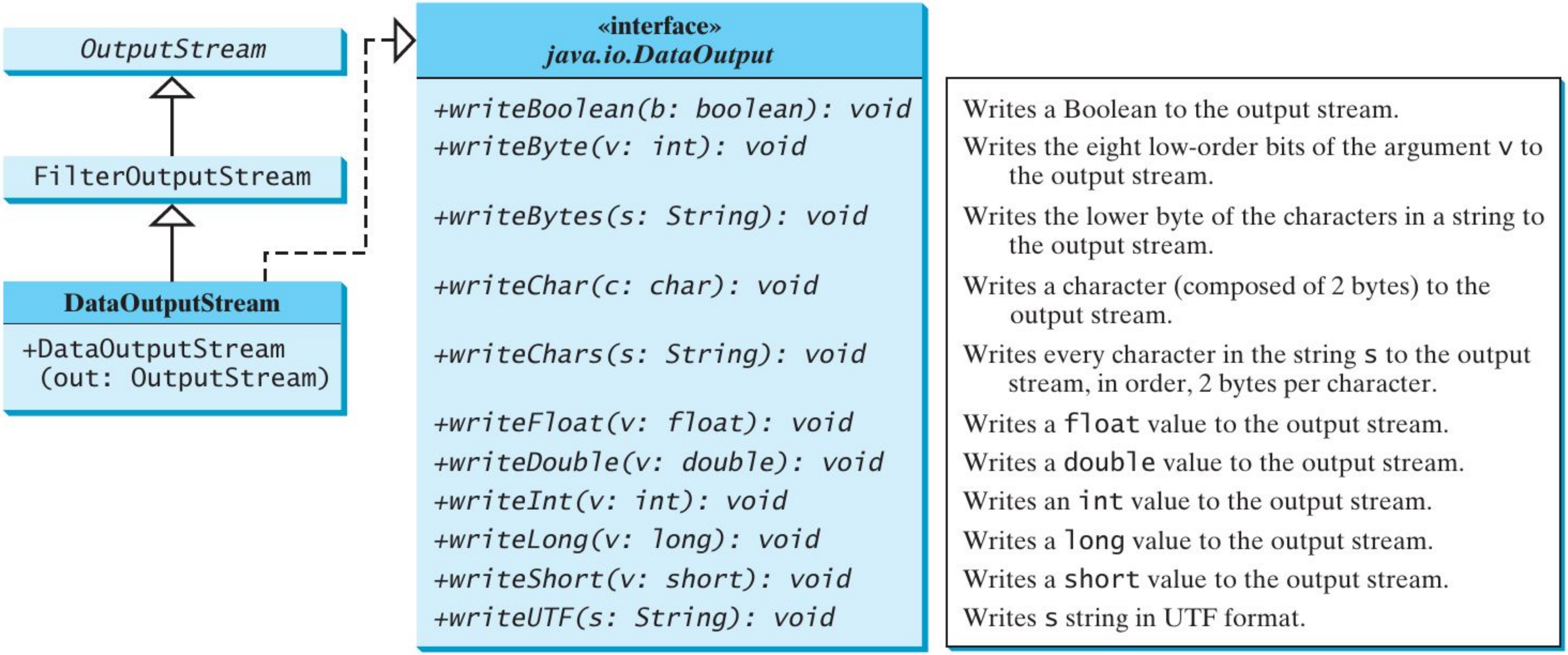
+DataInputStream(
 in: InputStream)

«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.

DataOutputStream



OutputStream

FilterOutputStream

DataOutputStream

+DataOutputStream
(out: OutputStream)

«interface»
java.io.DataOutput

+writeBoolean(b: boolean): void — Writes a Boolean to the output stream.

+writeByte(v: int): void — Writes the eight low-order bits of the argument `v` to the output stream.

+writeBytes(s: String): void — Writes the lower byte of the characters in a string to the output stream.

+writeChar(c: char): void — Writes a character (composed of 2 bytes) to the output stream.

+writeChars(s: String): void — Writes every character in the string `s` to the output stream, in order, 2 bytes per character.

+writeFloat(v: float): void — Writes a `float` value to the output stream.

+writeDouble(v: double): void — Writes a `double` value to the output stream.

+writeInt(v: int): void — Writes an `int` value to the output stream.

+writeLong(v: long): void — Writes a `long` value to the output stream.

+writeShort(v: short): void — Writes a `short` value to the output stream.

+writeUTF(s: String): void — Writes `s` string in UTF format.

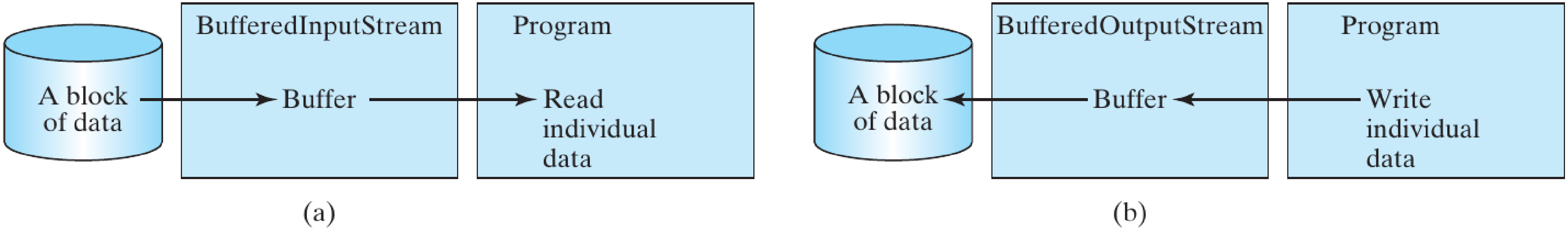
- See Code: **`TestDataStream.java`**

Detecting the End of a File

- If you keep reading at the end of an **`InputStream`**, an **`EOFException`** will be thrown.
- You can use this exception to detect the end of a file.
- See Code: **`DetectEndOfFile.java`**

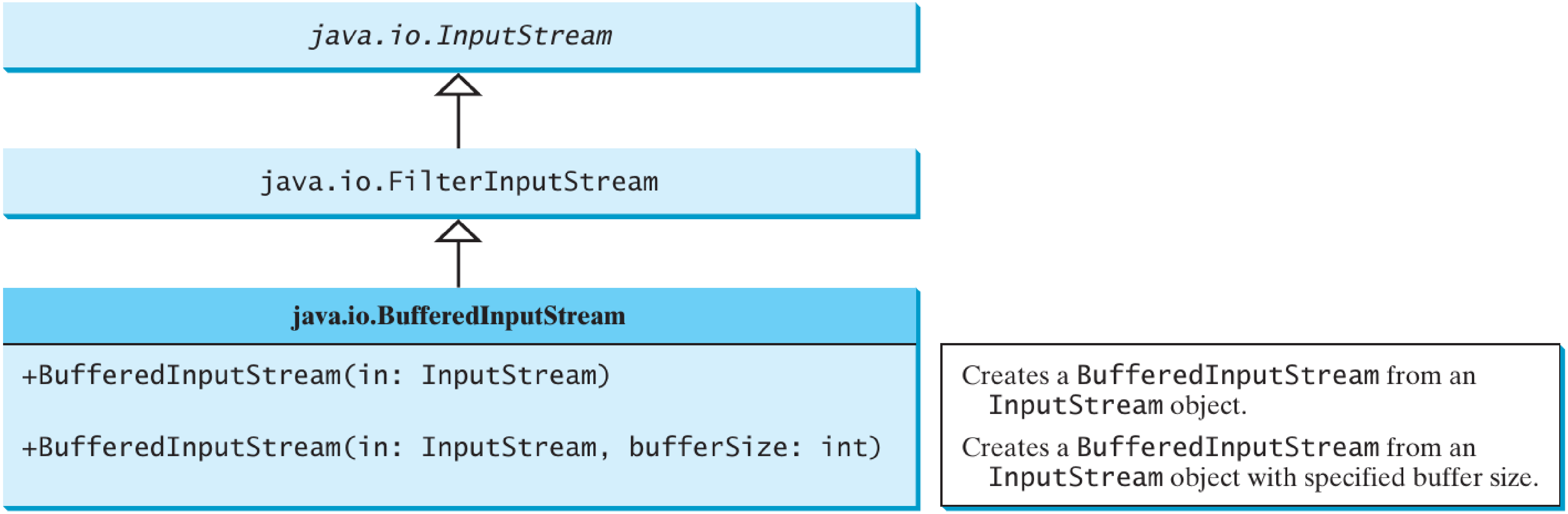
BufferedInputStream / BufferedOutputStream

- These classes can be used to speed up input and output by reducing the number of disk reads and writes.
- Generally a whole block of data is read / written into a buffer, and then transferred from the buffer to your program or output file.
- No new methods are introduced.
- The Constructors can take an optional buffer size, if no size is specified, the default size is 512 bytes.



Buffer I/O places data in a buffer for fast processing.

BufferedInputStream



java.io.InputStream

java.io.FilterInputStream

java.io.BufferedInputStream

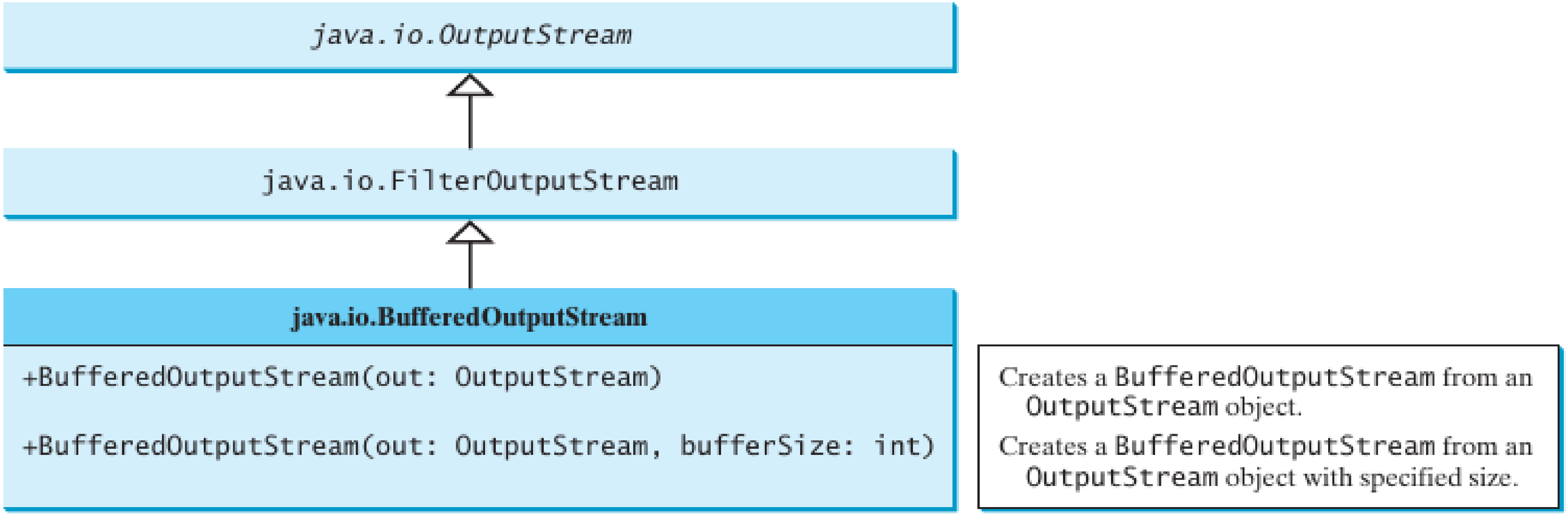
+BufferedInputStream(in: InputStream)

+BufferedInputStream(in: InputStream, bufferSize: int)

Creates a BufferedInputStream from an InputStream object.

Creates a BufferedInputStream from an InputStream object with specified buffer size.

BufferedOutputStream



java.io.OutputStream

java.io.FilterOutputStream

java.io.BufferedOutputStream

+BufferedOutputStream(out: OutputStream)

+BufferedOutputStream(out: OutputStream, bufferSize: int)

Creates a BufferedOutputStream from an OutputStream object.

Creates a BufferedOutputStream from an OutputStream object with specified size.

Case Study: Copying Files

Case Study: Copying Files

- This program lets a user make a copy of an input file and displays the number of bytes in the file.

- The program uses command line parameters in the following format:

```
java CopyFiles source_file target_file
```

- See Code: **`CopyFiles.java`**

Object I/O

ObjectInputStream / ObjectOutputStream

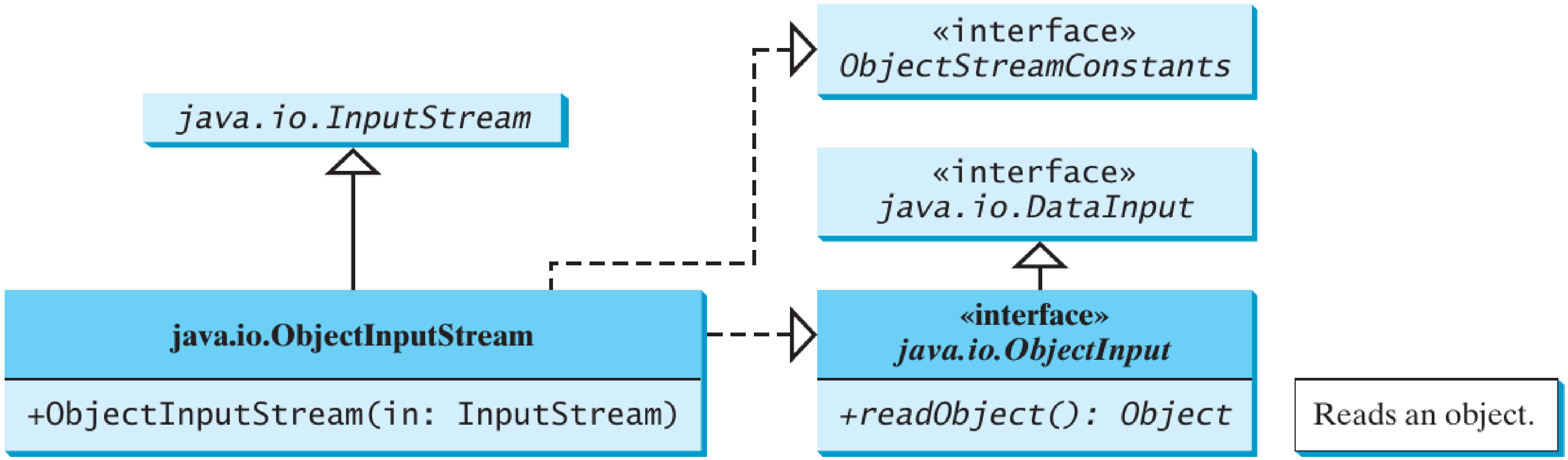
- **`ObjectInputStream`** / **`ObjectOutputStream`** classes can be used to read / write serializable objects.

- You can wrap either of these classes on any **`InputStream`** / **`OutputStream`**

- These classes can write entire objects, or they can write simple data types (because it inherits all the previous methods we have seen)
  - The data must be read back in the same order that it was written.

- Note: The **`readObject()`** method may throw a **`java.lang.ClassNotFoundException`** because when the JVM restores an object, it first loads the class of the object if the class was not previously loaded.

ObjectInputStream



«interface»
ObjectStreamConstants

java.io.InputStream

«interface»
java.io.DataInput

java.io.ObjectInputStream

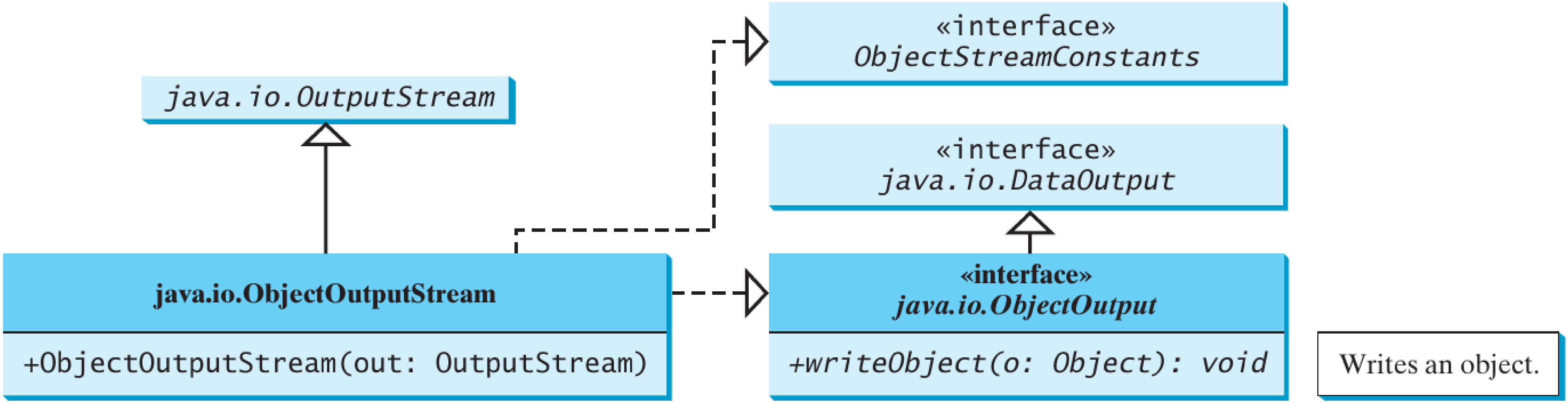
+ObjectInputStream(in: InputStream)

«interface»
java.io.ObjectInput

+readObject(): Object

Reads an object.

ObjectOutputStream



- See Code: **`ObjectIO.java`**

The `Serializable` Interface and `Serializable` Objects

- Not every object can be written to an output stream.

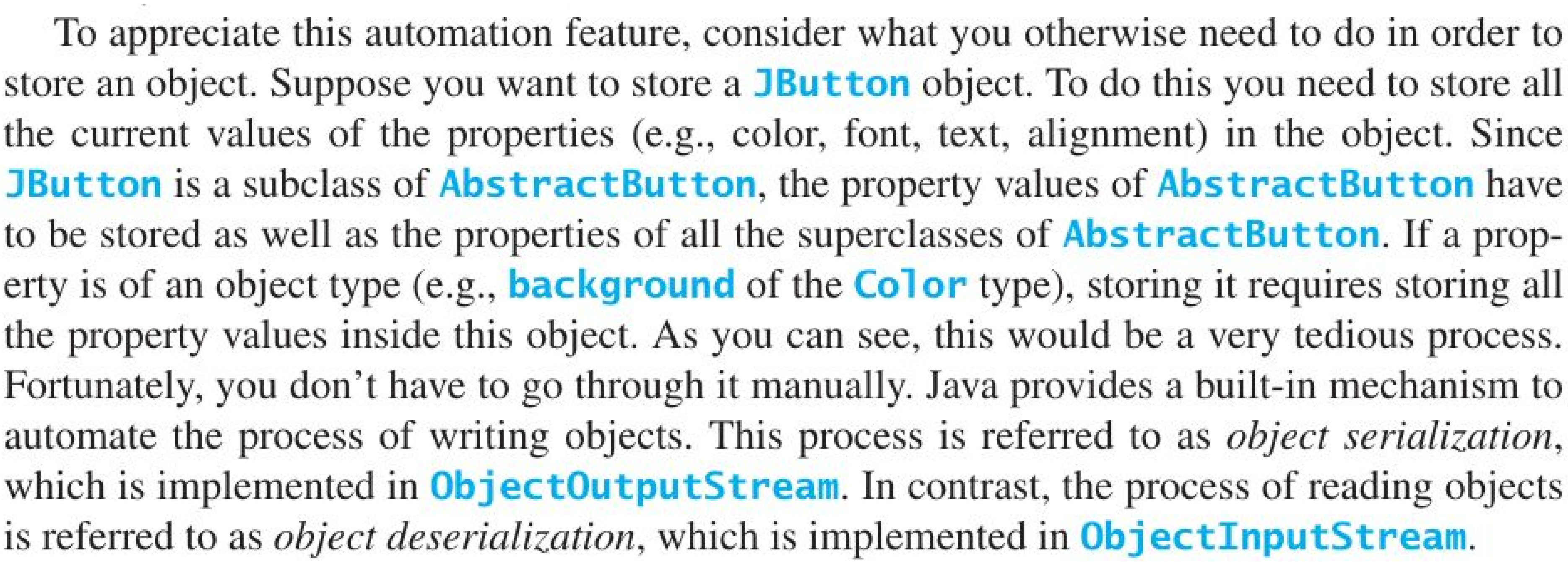
- Only ***serializable*** objects can be written
  - these objects are instances of the **`Serializable`** interface.
  - if you want to write an object to a file, its class MUST implement **`Serializable`**

- Simply put, just make sure your class implements Serializable.

- When serializable objects are stored, the class of the object is encoded
  - this includes class name, signature of class, values of the instance variables, closure of any other objects referenced by the object.
  - values of static variables are not stored.

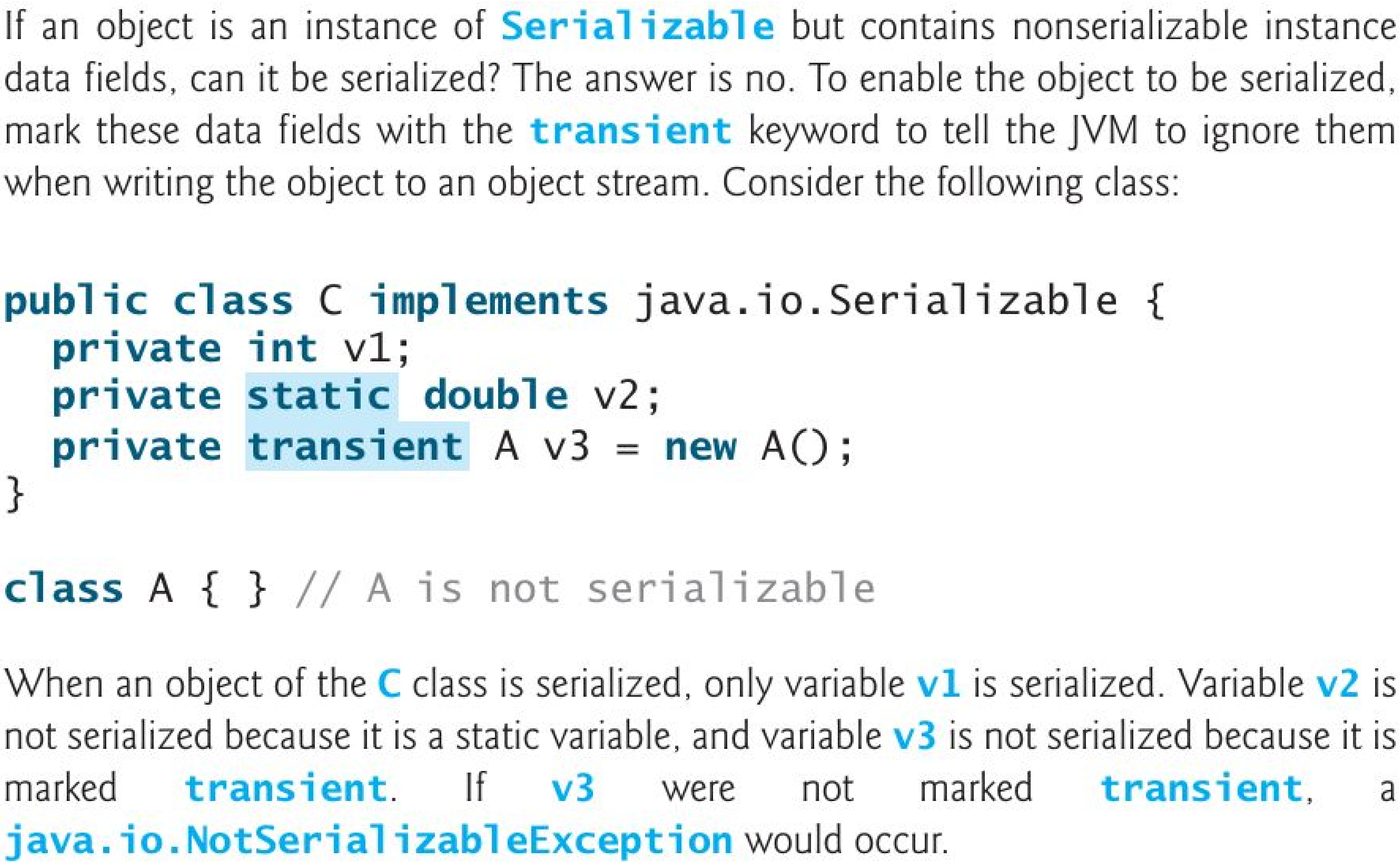
The `Serializable` Interface and `Serializable` Objects

- Why is this necessary?



To appreciate this automation feature, consider what you otherwise need to do in order to store an object. Suppose you want to store a `JButton` object. To do this you need to store all the current values of the properties (e.g., color, font, text, alignment) in the object. Since `JButton` is a subclass of `AbstractButton`, the property values of `AbstractButton` have to be stored as well as the properties of all the superclasses of `AbstractButton`. If a property is of an object type (e.g., `background` of the `Color` type), storing it requires storing all the property values inside this object. As you can see, this would be a very tedious process. Fortunately, you don't have to go through it manually. Java provides a built-in mechanism to automate the process of writing objects. This process is referred to as *object serialization*, which is implemented in `ObjectOutputStream`. In contrast, the process of reading objects is referred to as *object deserialization*, which is implemented in `ObjectInputStream`.

Nonserializable Fields



If an object is an instance of `Serializable` but contains nonserializable instance data fields, can it be serialized? The answer is no. To enable the object to be serialized, mark these data fields with the `transient` keyword to tell the JVM to ignore them when writing the object to an object stream. Consider the following class:

```
public class C implements java.io.Serializable {
  private int v1;
  private static double v2;
  private transient A v3 = new A();
}

class A { } // A is not serializable
```

When an object of the `C` class is serialized, only variable `v1` is serialized. Variable `v2` is not serialized because it is a static variable, and variable `v3` is not serialized because it is marked `transient`. If `v3` were not marked `transient`, a `java.io.NotSerializableException` would occur.

Example

- Suppose you want to send User object through a data stream (could be over a network too).

- You would probably want to expose data fields related to Name, Email, Address, Phone Number, but you would NOT want to expose a Password datafield.

- By making Password transient, you ensure that the data is not serialized and sent along with the object.

Duplicate Objects

- if the same object is written to an object stream more than once multiple copies are ***not*** stored.

- the first time an object is written a serial number is generated, the JVM writes the complete contents of the object along with the serial number

- if a copy of an object is written again then only the serial number is written.

- when the objects are read back, their references are the same since only one object is actually created in memory.

Serializing Arrays

- You can serialize an array if all of the elements in the array are serializable.
- Therefore, you can save an entire array into a file using the Object Streams.