Binary I/O

Text File vs. Binary File

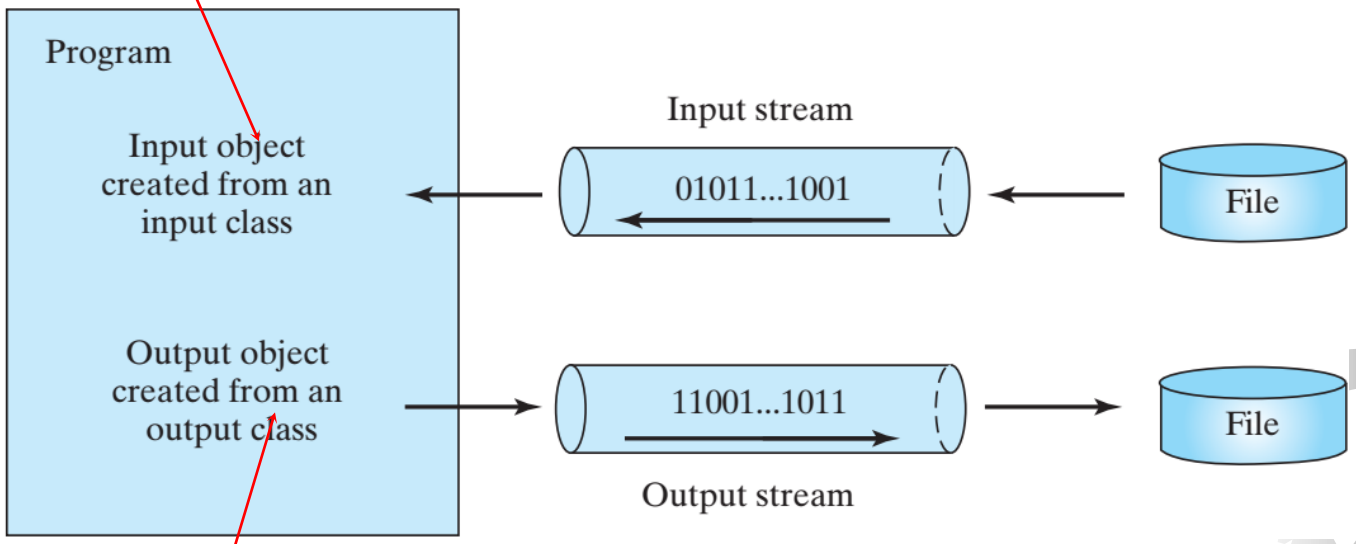
* you can imagine that
  - a text file consists of a sequence of characters
  - and a binary file consists of a sequence of bits.
* For example, the decimal integer 199:
  - in a text file: is stored as the sequence of three characters: '1', '9', '9'.
  - in a binary file: is stored as a byte-type value C7.

How is I/O Handled in Java?

* A File object encapsulates the properties of a file or a path, but does not contain the methods for reading/writing data from/to a file.

* To perform I/O, you should create objects using appropriate Java I/O classes.

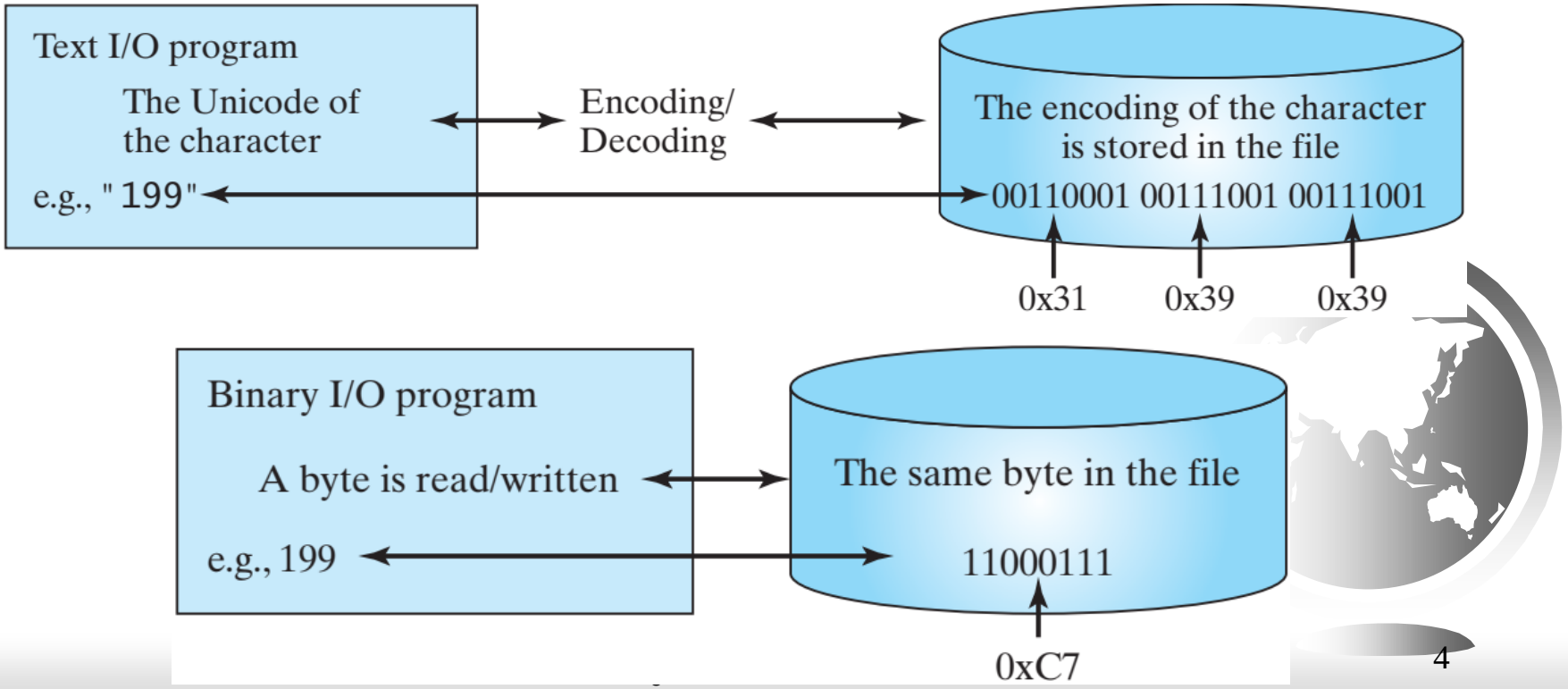
```
Scanner input = new Scanner(new File("temp.txt"));
System.out.println(input.nextLine());
```




```
PrintWriter output = new PrintWriter("temp.txt");
output.println("Java 101");
output.close();
```

Text I/O vs Binary I/O

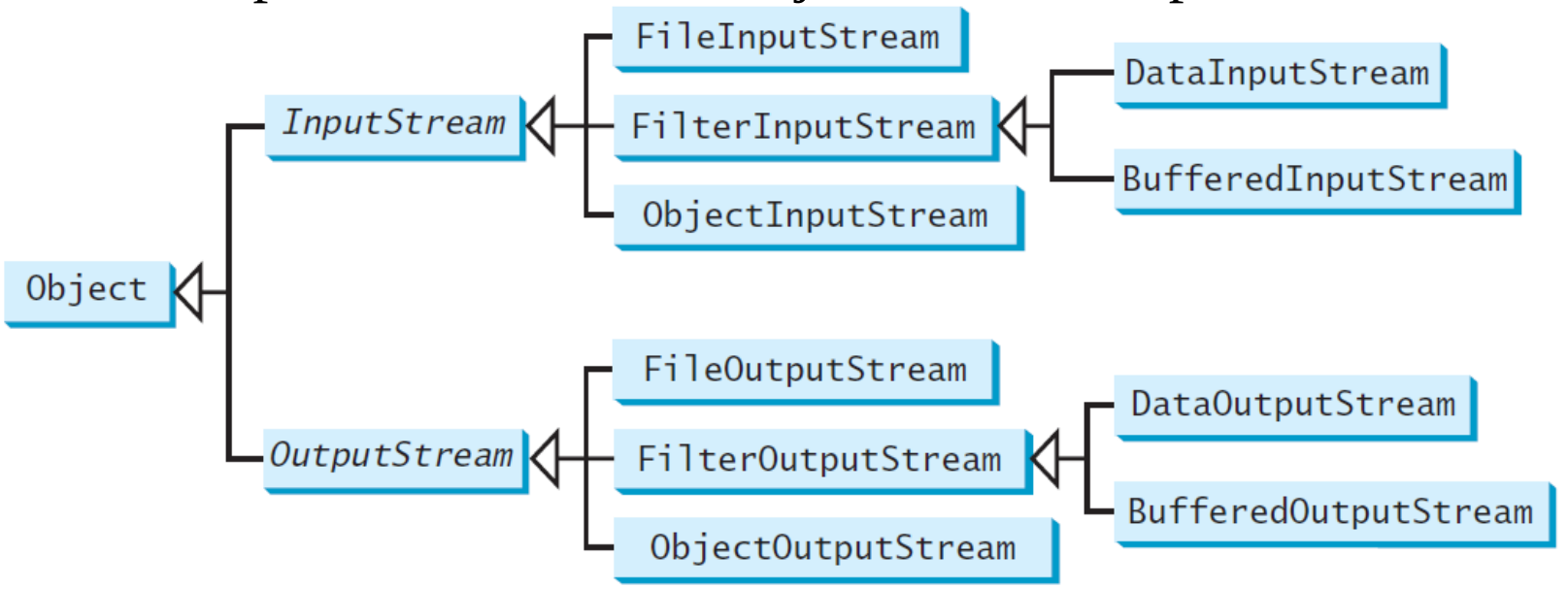
* Text I/O requires encoding and decoding. The JVM
  - converts a Unicode to a file specific encoding when writing a character
  - and coverts a file specific encoding to a Unicode when reading a character.
* Binary I/O does not require conversions.
  - When you write a byte to a file, the original byte 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

* The design of the Java I/O classes is a good example of applying inheritance, where
  - common operations are generalized in superclasses,
  - and subclasses provide specialized operations.
  - InputStream is the root for binary input classes,
  - and OutputStream is the root for binary output classes.

* Note: All the methods in the binary I/O classes are declared to throw java.io.IOException or a subclass of java.io.IOException.



InputStream

* The abstract InputStream class defines the methods for the input stream of bytes.

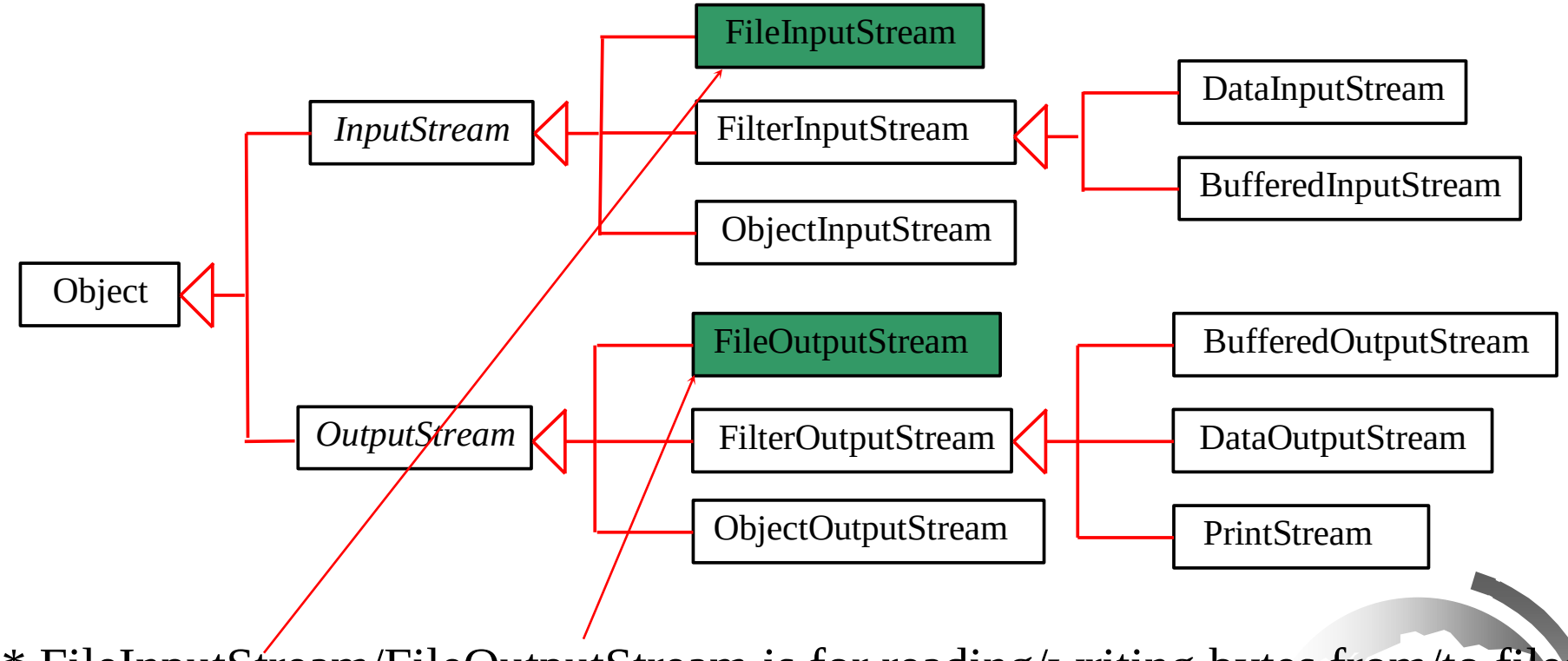
java.io.InputStream	
+read(): int	Reads the next byte of data from the input stream. The value byte is returned as an `int` value in the range `0` to `255`. If no byte is available because the end of the stream has been reached, the value –1 is returned.
`+read(b: byte[]): int`	Reads up to `b.length` 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

* The abstract OutputStream class defines the methods for the output stream of bytes.

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 them are inherited from their superclasses.
* FileInputStream/FileOutputStream does not introduce new methods.

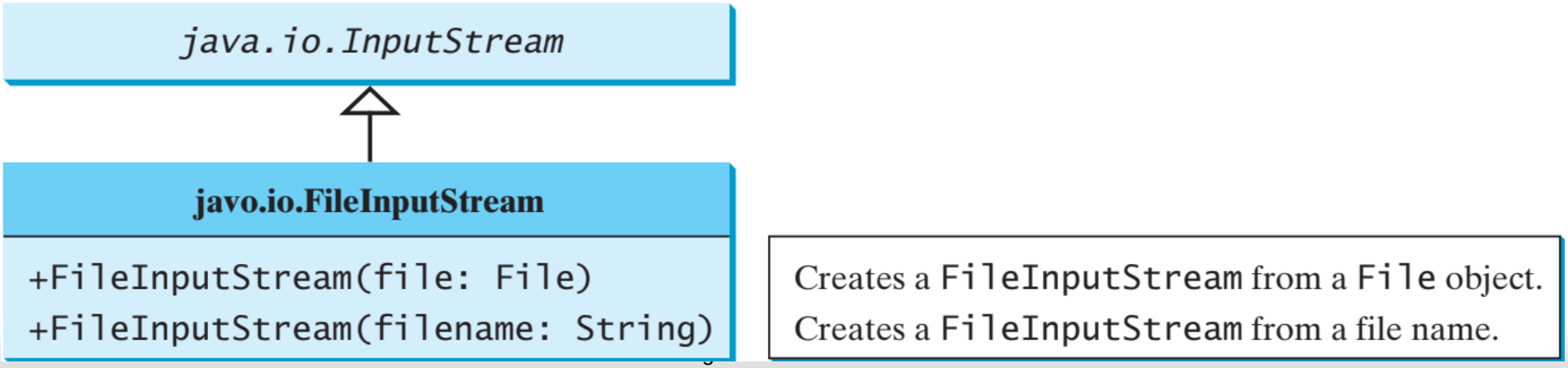
FileInputStream

* 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)

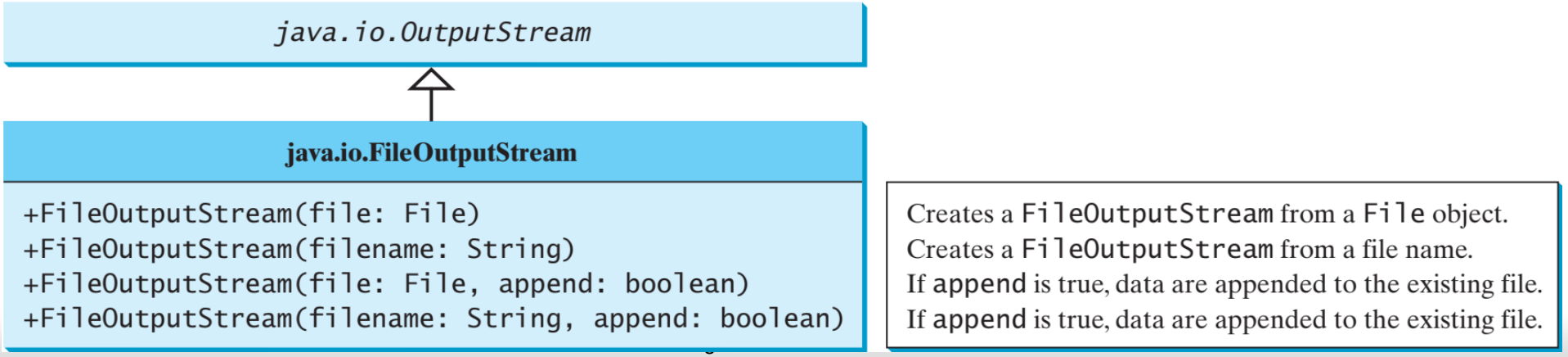
  public FileOutputStream(File file, boolean append)

* 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 retain the current content and append new data into the file, use the last two constructors by passing true to the append parameter.



java.io.IOException

* Almost all the methods in 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 trycatch block.

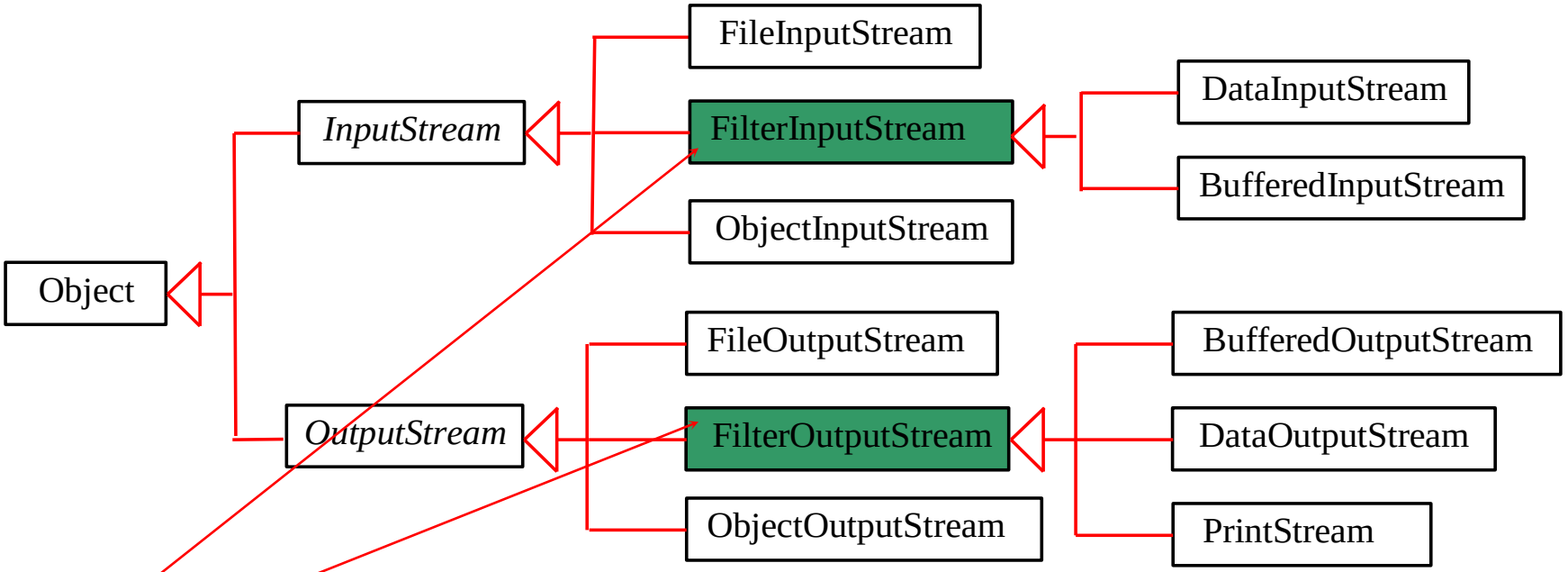
Declaring exception in the method

```
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();
  }
}
```

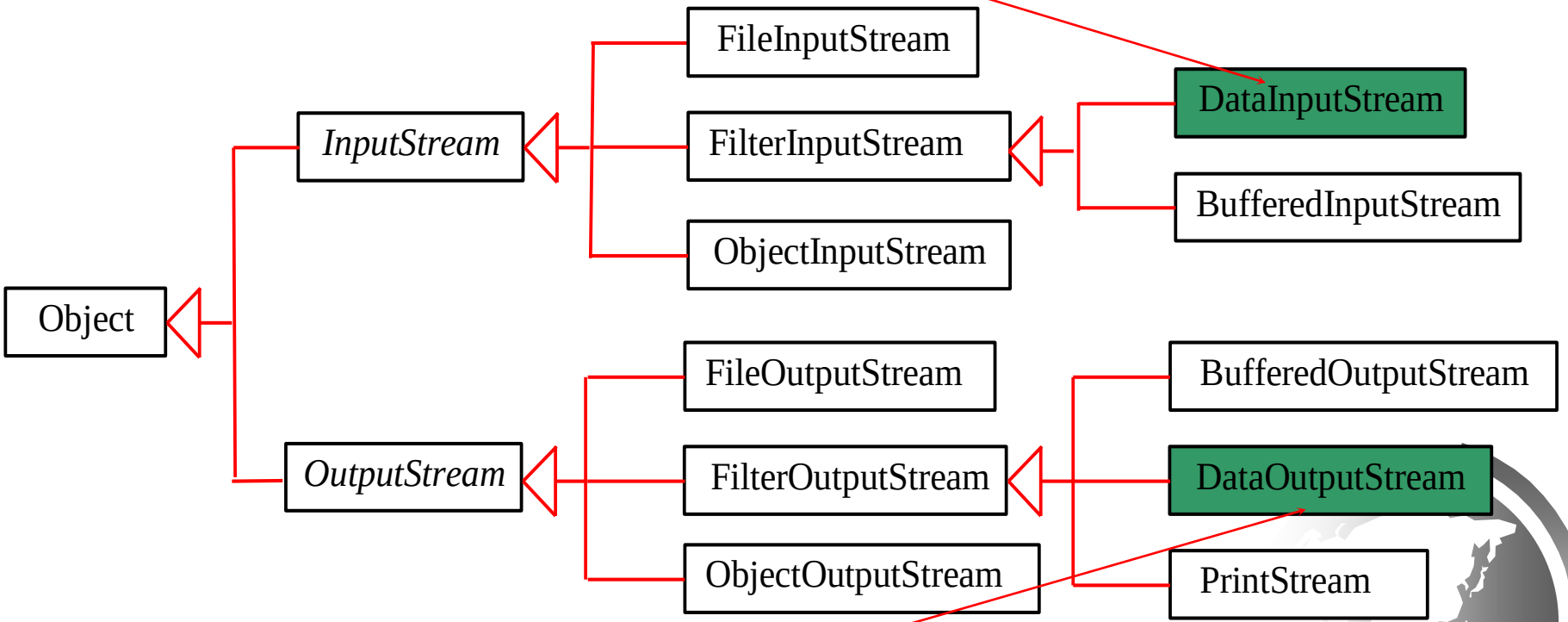
FilterInputStream/FilterOutputStream



* *Filter streams* are streams that filter bytes for some purpose.
  - The basic byte input stream can only be used for reading bytes.
  - To read integers, doubles, or strings, you need a filter class to wrap the byte input stream.
  - 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

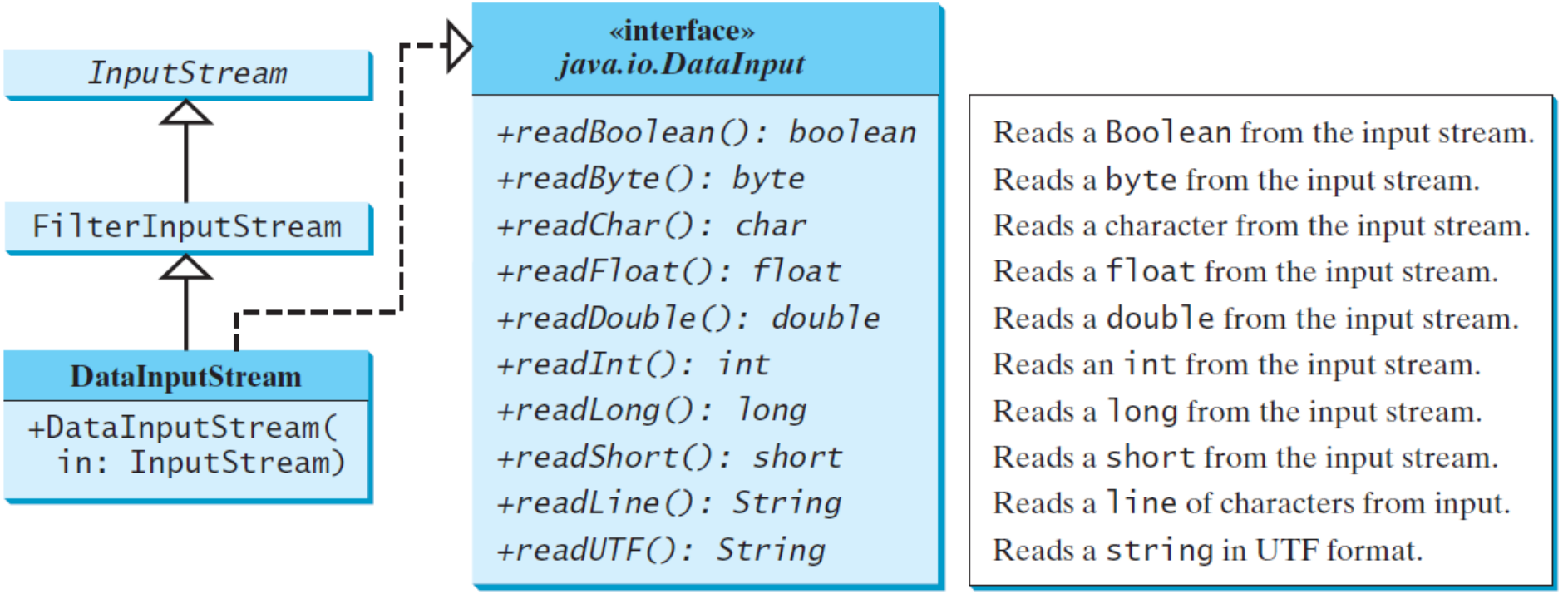
DataInputStream reads bytes from the stream and converts them into appropriate primitive type values or strings.



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

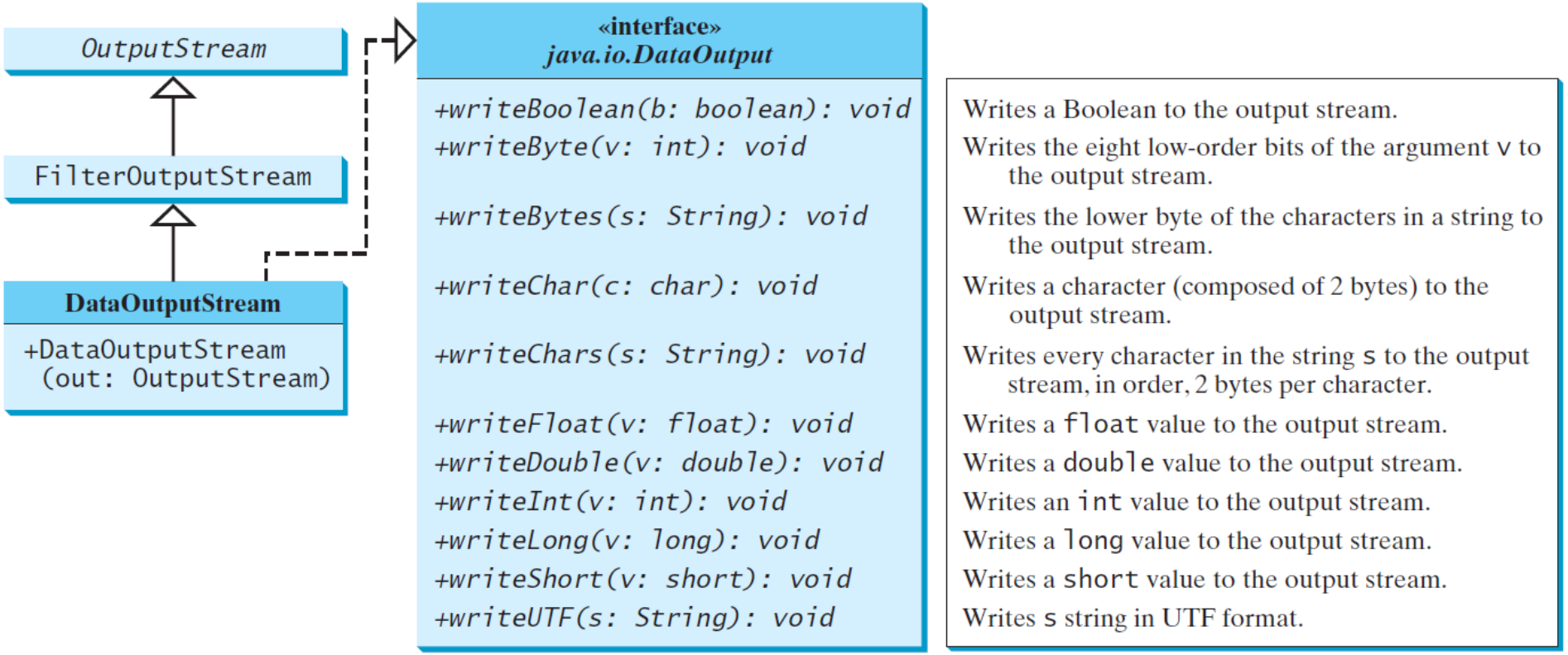
DataInputStream

* DataInputStream extends FilterInputStream and implements the DataInput interface.



DataOutputStream

* DataOutputStream extends FilterOutputStream and implements the DataOutput interface.



Characters and Strings in Binary I/O

* A Unicode character consists of two bytes.

* The writeChar(char c) method writes the Unicode of character c to the output.

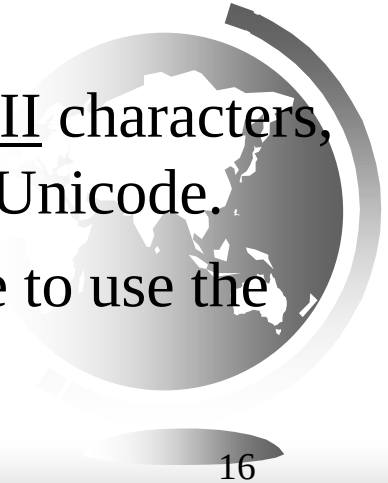
* The writeChars(String s) method writes the Unicode for each character in the string s to the output.

* The writeBytes(String s) method writes the lower byte of the Unicode for each character in the string s to the output.

  - The high byte of the Unicode is discarded.

  - The method is suitable for strings that consist of ASCII characters, since an ASCII code is stored only in the lower byte of a Unicode.

  - If a string consists of non-ASCII characters, you have to use the writeChars method to write the string.



Characters and Strings in Binary I/O

Why UTF-8? What is UTF-8?

* UTF-8 is a coding scheme that allows systems to operate with both ASCII and Unicode efficiently.

- Most operating systems use ASCII. Java uses Unicode. The ASCII character set is a subset of the Unicode character set.

- Most applications need only the ASCII character set, so it is a waste to represent an 8-bit ASCII character as a 16-bit Unicode character.

- The UTF-8 is an alternative scheme that stores a character using 1, 2, or 3 bytes.

  - ASCII values (less than 0x7F) are coded in one byte.
  - Unicode values less than 0x7FF are coded in two bytes.
  - Other Unicode values are coded in three bytes.

* The writeUTF(String s) method converts a string into a series of bytes in the UTF-8 format and writes them into an output stream. The readUTF() method reads a string that has been written using writeUTF.

Using DataInputStream/DataOutputStream

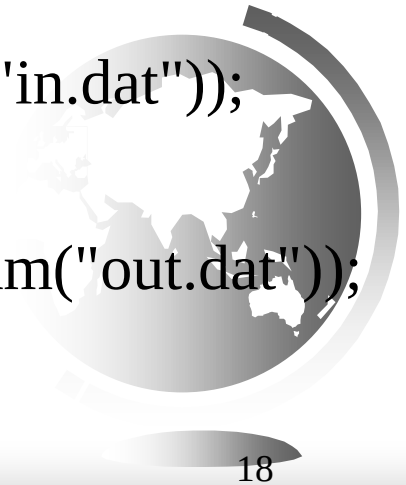
* Data streams are used as wrappers on existing input and output streams to filter data in the original stream.

* They are created using the following constructors:

```
public DataInputStream(InputStream instream)
public DataOutputStream(OutputStream outstream)
```

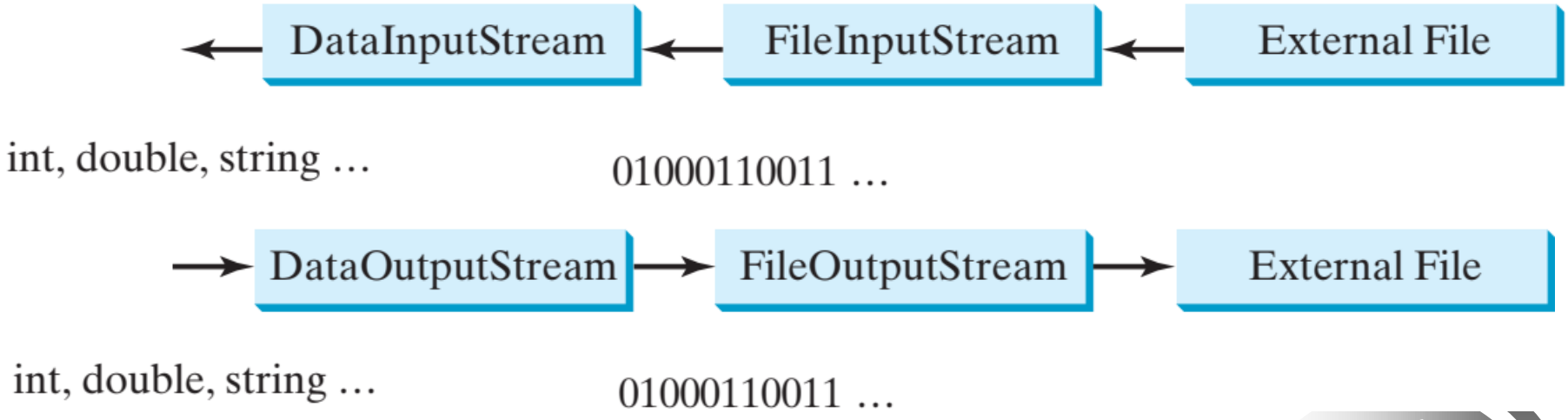
* The statements given below create data streams.

```
DataInputStream infile =
    new DataInputStream(new FileInputStream("in.dat"));
DataOutputStream outfile =
    new DataOutputStream(new FileOutputStream("out.dat"));
```



Using DataInputStream/DataOutputStream

* You can view DataInputStream/FileInputStream and DataOutputStream/FileOutputStream working in a pipe line as shown.



CAUTION: You have to read the data in the same order and same format in which they are stored.

Checking End of File

TIP:

* If you keep reading data at the end of a stream, an EOFException would occur.

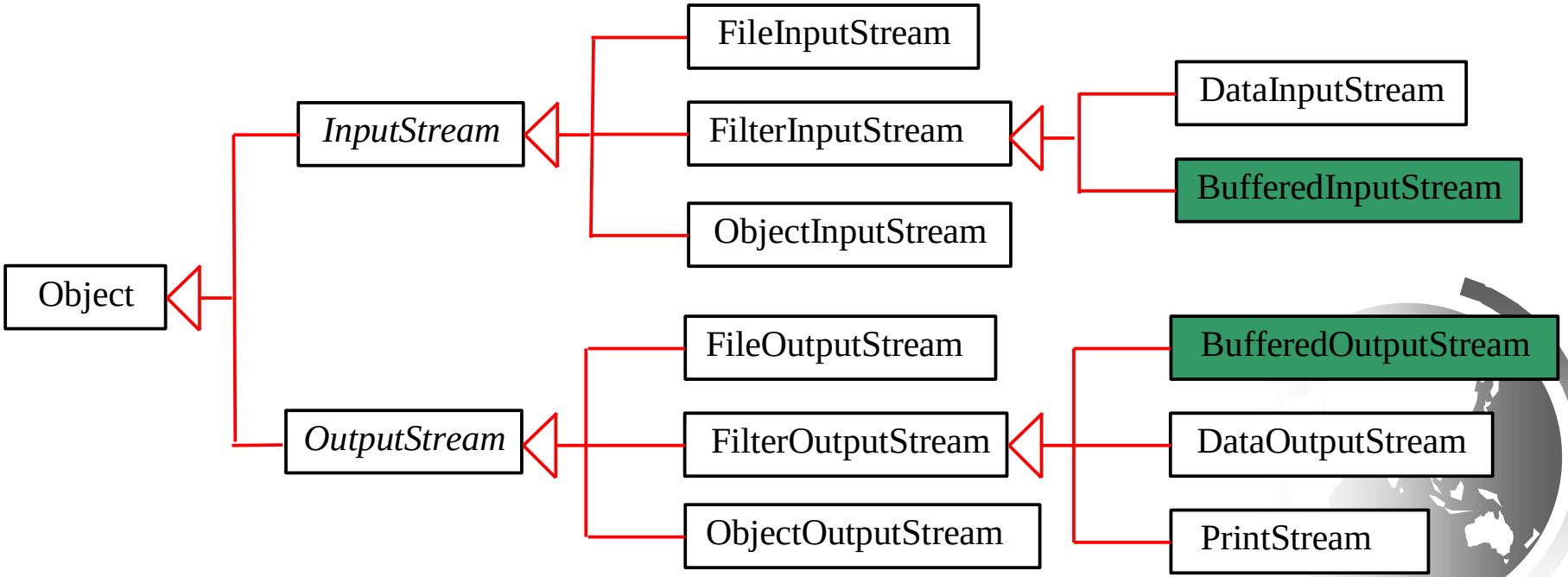
* So how do you check the end of a file?

  - You can use input.available() to check it.

  - input.available() == 0 indicates that it is the end of a file.

BufferedInputStream/BufferedOutputStream

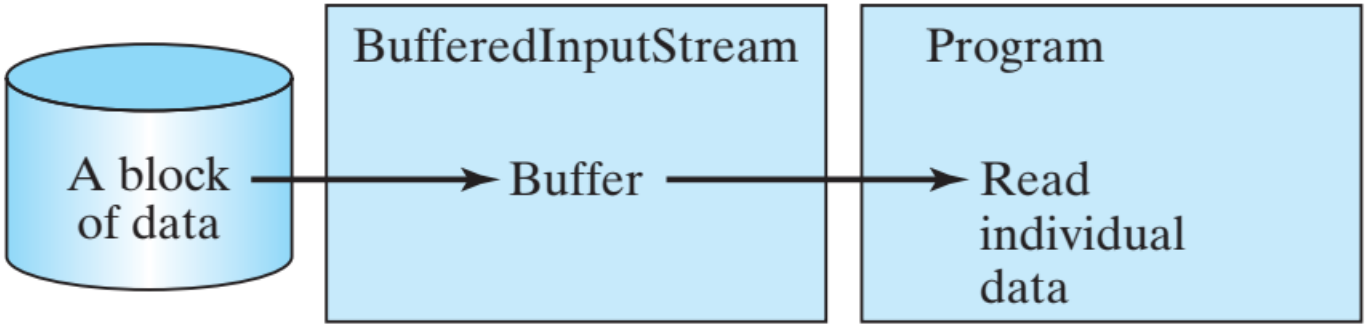
* BufferedInputStream/BufferedOutputStream can be used to speed up input and output by reducing the number of disk reads and writes.



BufferedInputStream/BufferedOutputStream

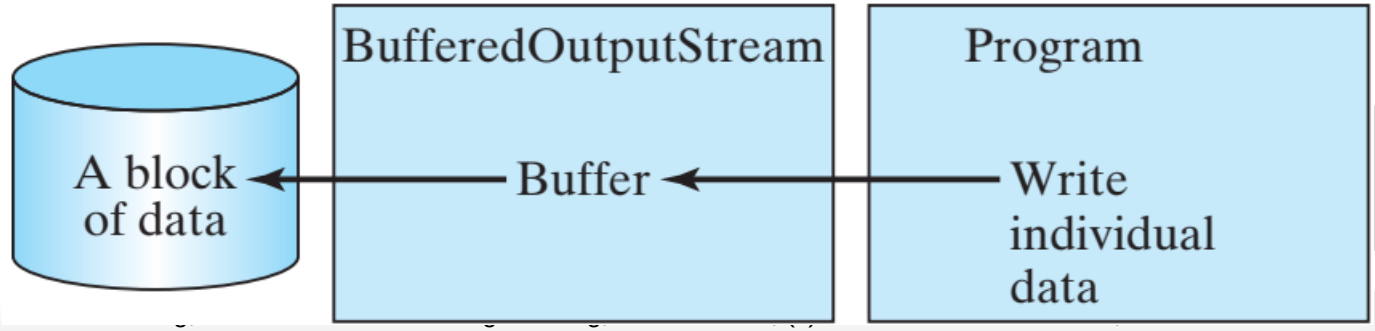
* Using BufferedInputStream, the whole block of data on the disk is read into the buffer in the memory once.

* The individual data are then delivered to the program from the buffer.



* Using BufferedOutputStream, the individual data are first written to the buffer in the memory.

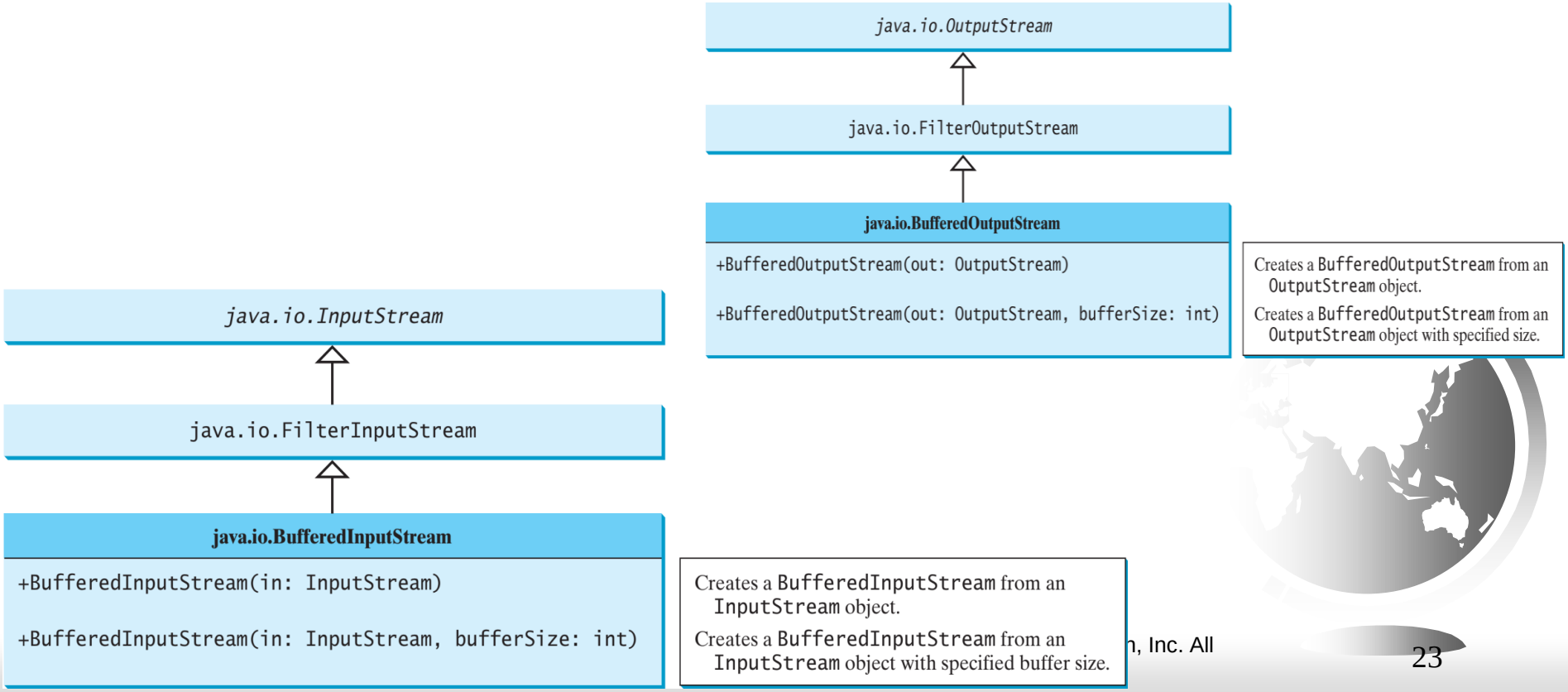
* When the buffer is full, all data in buffer are written to the disk once.



BufferedInputStream/BufferedOutputStream

* BufferedInputStream/BufferedOutputStream does not contain new methods.

  - All the methods in BufferedInputStream/BufferedOutputStream are inherited from the InputStream/OutputStream classes.



Constructing BufferedInputStream/BufferedOutputStream

```
// Create a BufferedInputStream
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int bufferSize)

// Create a BufferedOutputStream
public BufferedOutputStream(OutputStream out)
public BufferedOutputStream(OutputStream out, int bufferSize)
```

Tip:

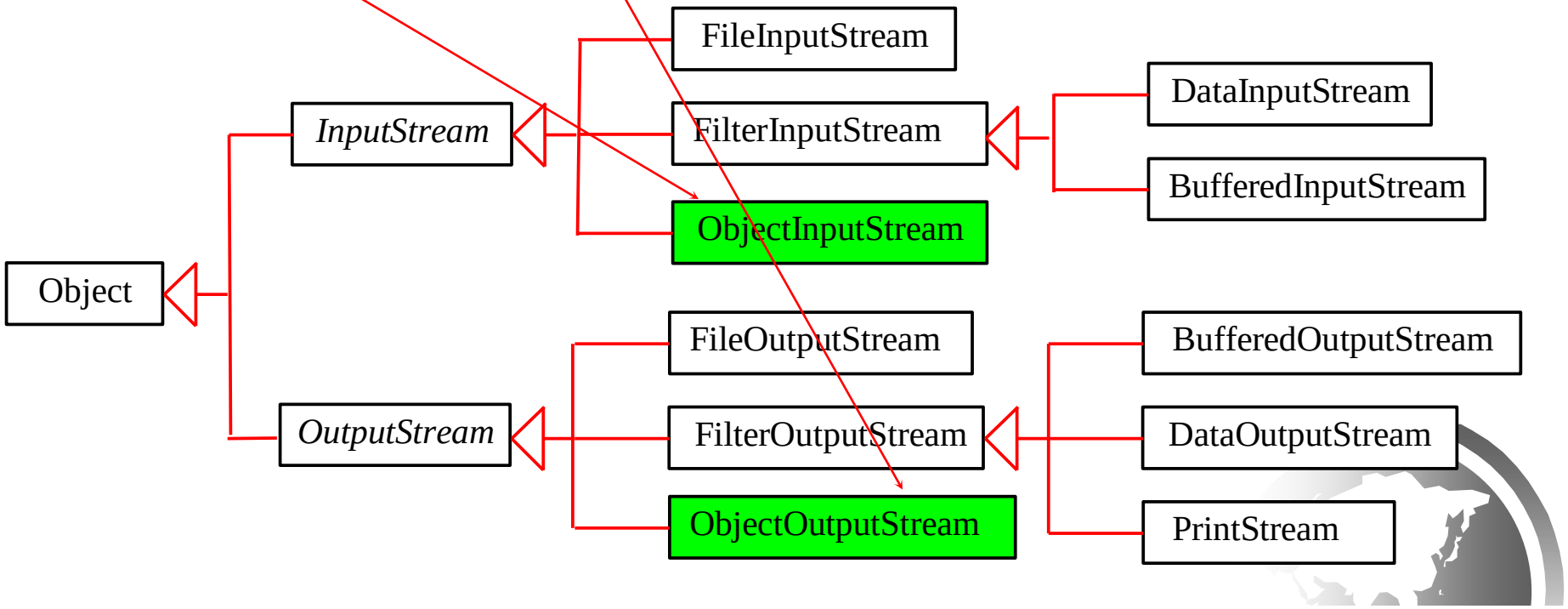
* You should always use buffered I/O to speed up input and output.
  - For small files, you may not notice performance improvements.
  - However, for large files—over 100 MB—you will see substantial improvements.

Optional

Object I/O

* DataInputStream/DataOutputStream enables you to perform I/O for primitive type values and strings.

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

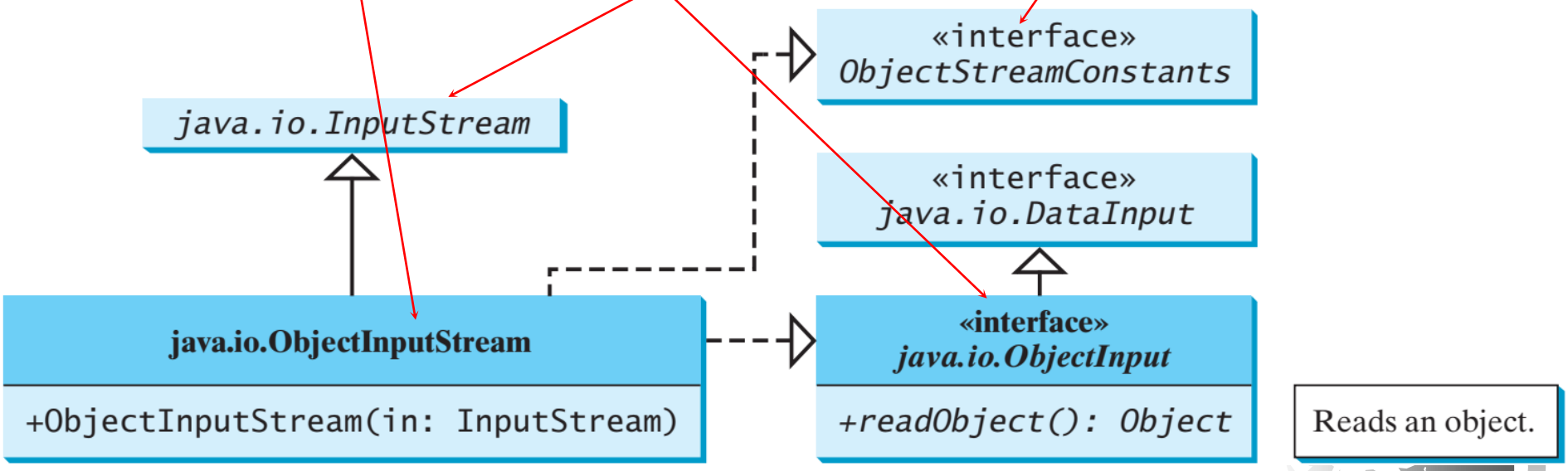


* Since ObjectInputStream/ObjectOutputStream contains all the functions of DataInputStream/DataOutputStream,

* you can replace DataInputStream/DataOutputStream completely with ObjectInputStream/ObjectOutputStream.

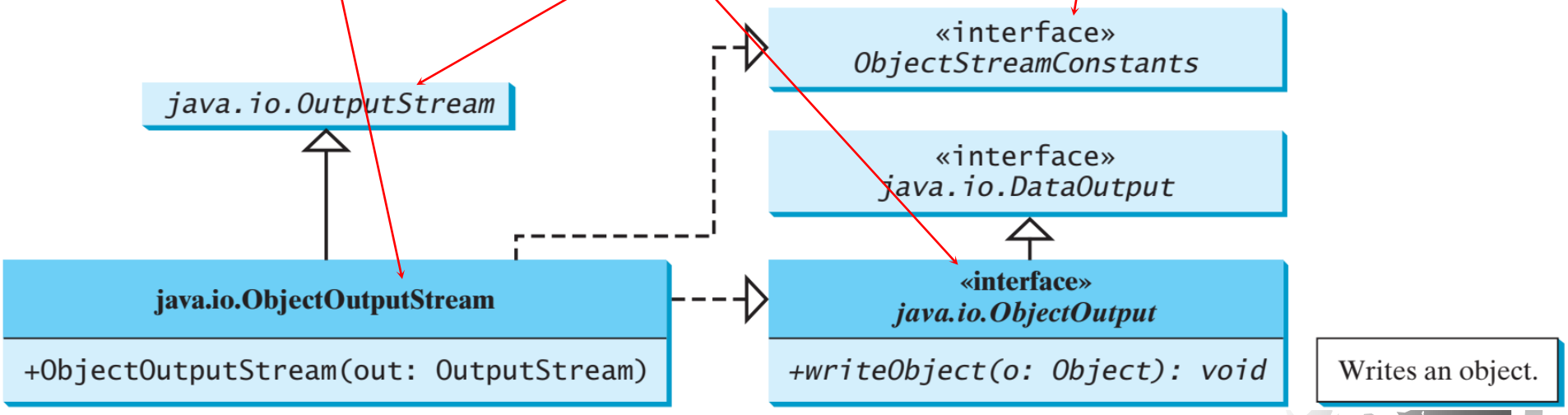
ObjectInputStream

* ObjectInputStream extends InputStream and implements ObjectInput and ObjectStreamConstants.



ObjectOutputStream

* ObjectOutputStream extends OutputStream and implements ObjectOutput and ObjectStreamConstants.



Using Object Streams

* You may wrap an ObjectInputStream/ObjectOutputStream on any InputStream/OutputStream using the following constructors:

// Create an ObjectOutputStream

public ObjectOutputStream(OutputStream out)

```
import java.io.*;

public class TestObjectOutputStream {
public static void main(String[] args) throws IOException {
  try ( // Create an output stream for file object.dat
    ObjectOutputStream output =
      new ObjectOutputStream(new FileOutputStream("object.dat"));
  ) {
    // Write a string, double value, and object to the file
    output.writeUTF("John");
    output.writeDouble(85.5);
    output.writeObject(new java.util.Date());
  }
}
}
```

Using Object Streams

* You may wrap an ObjectInputStream/ObjectOutputStream on any InputStream/OutputStream using the following constructors:

// Create an ObjectInputStream

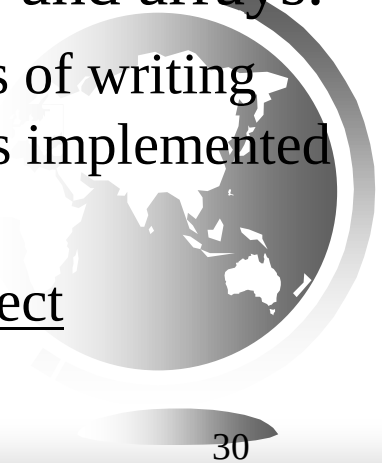
public ObjectInputStream(InputStream in)

```
import java.io.*;                          John 85.5 Sun Dec 04 10:35:31 EST 2011

public class TestObjectInputStream {
  public static void main(String[] args)
    throws ClassNotFoundException, IOException {
    try ( // Create an input stream for file object.dat
      ObjectInputStream input =
        new ObjectInputStream(new FileInputStream("object.dat"));
    ) {
      // Read a string, double value, and object from the file
      String name = input.readUTF();
      double score = input.readDouble();
      java.util.Date date = (java.util.Date)(input.readObject());
      System.out.println(name + " " + score + " " + date);
    }
  }
}
```

The `Serializable` Interface

* Not all objects can be written to an output stream.
  - Objects that can be written to an object stream is said to be *serializable*.
  - A serializable object is an instance of the java.io.Serializable interface.
  - So the class of a serializable object must implement Serializable.
* The Serializable interface is a marker interface.
  - It has no methods,
  - you don't need to add additional code that implements Serializable.
* Implementing this interface enables the Java serialization mechanism to automate the process of storing objects and arrays.
  - Java provides a built-in mechanism to automate the process of writing objects. This process is referred as object serialization, which is implemented in ObjectOutputStream.
  - In contrast, the process of reading objects is referred as object deserialization, which is implemented in ObjectInputStream.



The `Serializable` Interface, cont.

* Many classes in the Java API implement Serializable.

- All the wrapper classes for primitive type values, java.math.BigInteger, java.math.BigDecimal, java.lang.String,java.lang.StringBuilder, java.lang.StringBuffer, java.util.Date, and java.util.ArrayList implement java.io.Serializable.

- Attempting to store an object that does not support the Serializable interface would cause a NotSerializableException.

When a serializable object is stored, the class of object is encoded

- this includes the class name and the signature of the class, the values of the object's instance variables, and the closure of any other objects referenced by the object.

- The values of the object's static variables are not stored.

The `transient` Keyword

* If an object is an instance of Serializable, but it contains non-serializable instance data fields, can the object be serialized?

* The answer is no.

* To enable the object to be serialized,

  - you can use the transient keyword to mark these data fields to tell the JVM to ignore these fields when writing the object to an object stream.

The `transient` Keyword, cont.

* Consider the following class:

```
public class Foo 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 Foo 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.

Serializing Arrays

* An array is serializable if all its elements are serializable.

* So an entire array can be saved using writeObject into a file and later restored using readObject.

* Here is an example that stores an array of five int values and an array of three strings, and reads them back to display on the console.

Random Access Files

* sequential-access file: A file that is opened using a sequential stream.
  - The contents of a sequential-access file can't be updated.
  - However, it is often necessary to modify files.

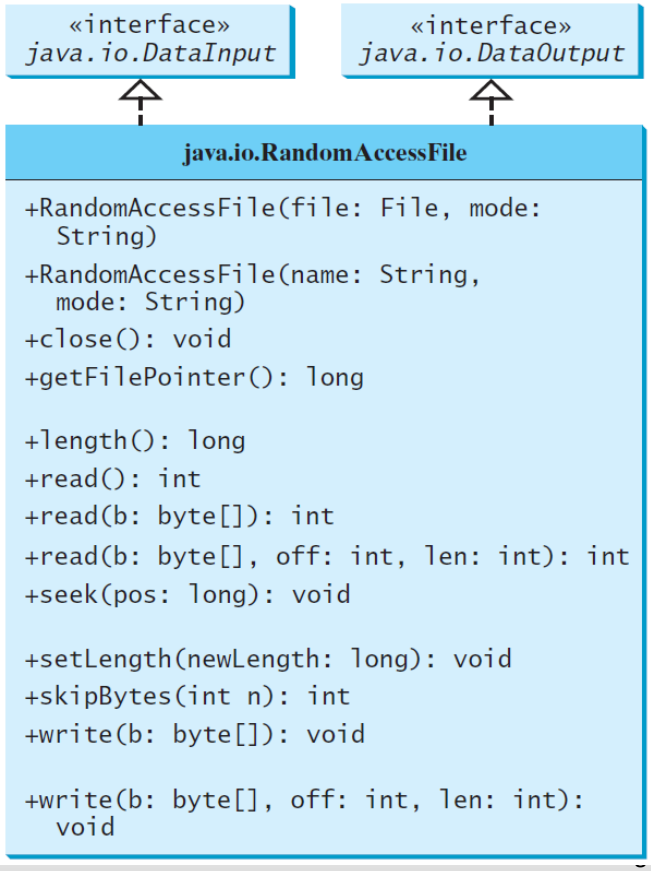
* Java provides the RandomAccessFile class
  - to allow data to be read from and written to at any locations in a file.
  - A file that is opened using the RandomAccessFile class is known as a random-access file

RandomAccessFile

* The RandomAccessFile class implements two interfaces.

- The DataInput interface: defines the methods for reading primitive-type values and strings (e.g., readInt, readDouble, readChar, readBoolean, readUTF)

- the DataOutput interface: defines the methods for writing primitive-type values and strings (e.g., writeInt, writeDouble, writeChar, writeBoolean, writeUTF).



+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 number of bytes in 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.

RandomAccessFile, cont.

* When creating a RandomAccessFile, you can specify one of two modes: r or rw.

- Mode r means that the stream is read-only, and mode rw indicates that the stream allows both read and write.

- For example:

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

- If test.dat already exists, raf is created to access it;
- if test.dat does not exist, a new file named test.dat is created, and raf is created to access the new file.
- The method raf.length() returns the number of bytes in test.dat at any given time.
- If you append new data into the file, raf.length() increases.

File Pointer

* A random access file consists of a sequence of bytes.

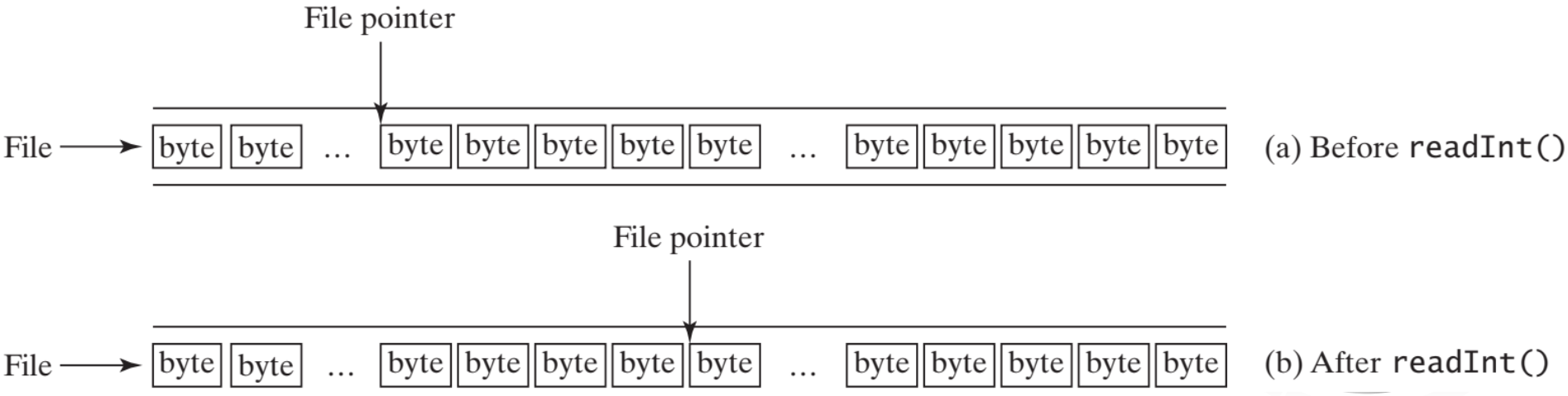
* There is a special marker called *file pointer* that is positioned at one of these bytes.

  - A read or write operation takes place at the location of the file pointer.

  - When a file is opened, the file pointer sets at the beginning of the file.

  - When you read or write data to the file, the pointer moves forward to the next data.

  - For example, if you read an int value using readInt(), the JVM reads four bytes from the file pointer and now the pointer is four bytes ahead of the previous location.



`RandomAccessFile` Methods

* Many methods in `RandomAccessFile` are the same as those in `DataInputStream` and `DataOutputStream`.

* For example,

  `readInt()`, `readLong()`, `writeDouble()`, `readLine()`, `writeInt()`, and `writeLong()`

  can be used in data input stream or data output stream as well as in `RandomAccessFile` streams.

`RandomAccessFile` Methods, cont.

```
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() 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

```
RandomAccessFile raf =
  new RandomAccessFile("test.dat", "rw");
  // allows read and write


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