

Binary File I/O

Introduction to Programming and
Computational Problem Solving:
Accelerated Pace

CSE 11

Lecture 18

Announcements

- Assignment 8 is due today, 11:59 PM
 - Upgrade beginning Jun 8, 12:01 AM
- Please complete Student Evaluations of Teaching (SET)
- Please complete TA and tutor evaluations
- Final assessments are Jun 11-14
- Assignments 5-8 upgrades due Jun 12, 11:59 PM

Files

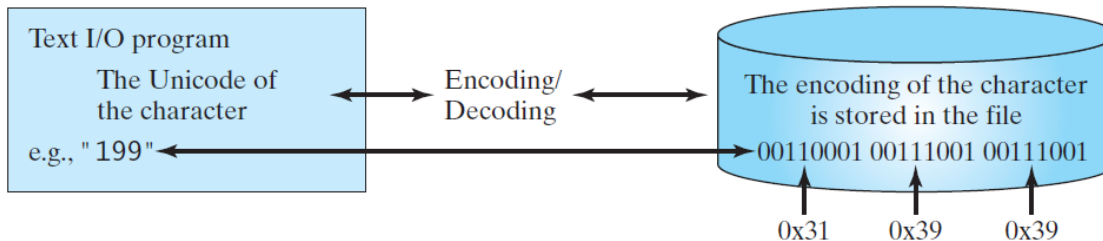
- Files can be classified as either text or binary
 - Human readable files are **text files**
 - All other files are **binary files**
- Java provides many classes for performing text file Input/Output (I/O) and binary file I/O

File I/O

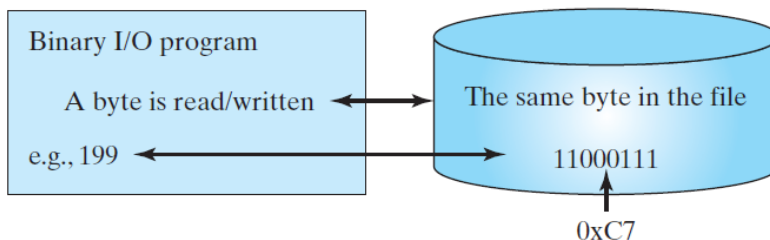
- Remember, a `File` object encapsulates the properties of a file (or directory), *but does not contain the methods for reading/writing data from/to a file*
- In order to perform I/O, you need to create objects using appropriate Java I/O classes
 - The objects contain the methods for reading/writing data from/to a file
- Text file I/O
 - Use the `Scanner` class for reading text data from a file
 - The JVM converts a file specific encoding to Unicode when reading a character
 - Use the `PrintWriter` class for writing text data to a file
 - The JVM converts Unicode to a file specific encoding when writing a character

Binary file I/O

- Binary file I/O does not involve encoding or decoding and thus is more efficient than text file I/O
- Binary files are independent of the encoding scheme on the host machine



(a)

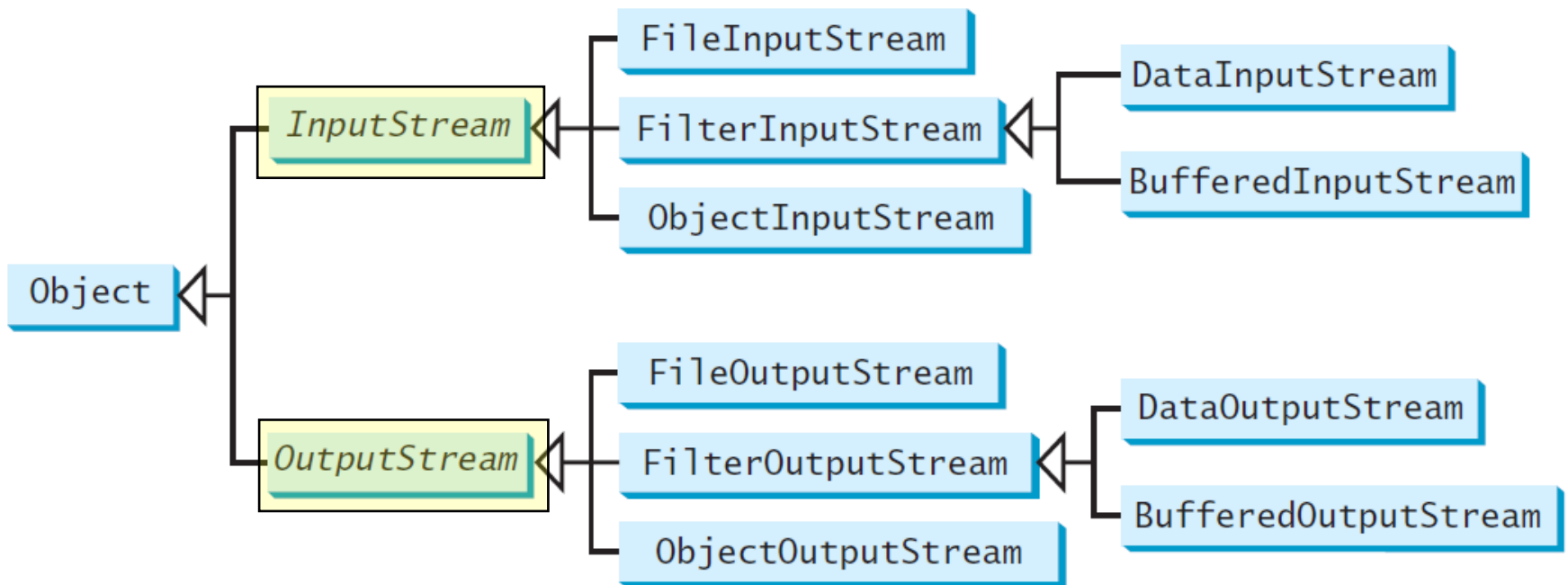


(b)

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 abstract `InputStream` is the root class for reading binary data
- The abstract `OutputStream` is the root class for writing binary data



The InputStream class

<https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html>

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/InputStream.html>

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

The OutputStream class

<https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html>

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/OutputStream.html>

The value is a byte as an int type

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

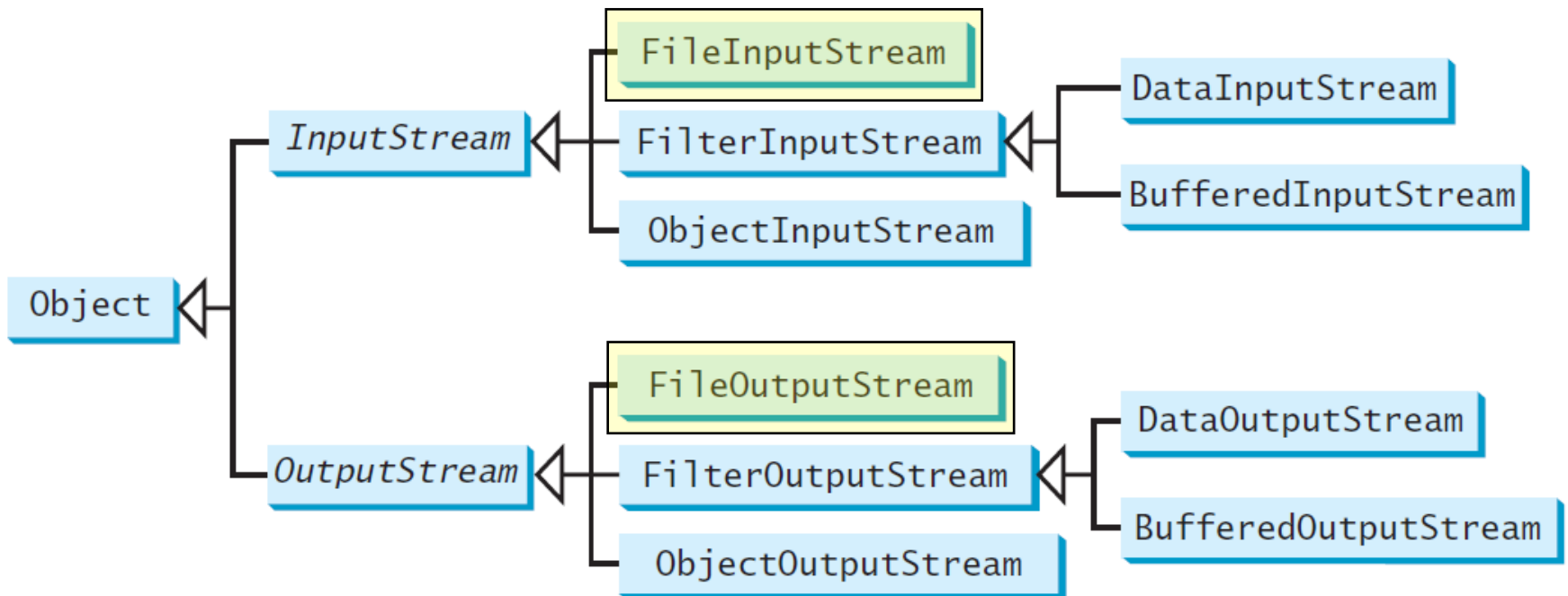
Closes this output stream and releases any system resources associated with the stream.

+flush(): void

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

Binary file I/O classes

- `FileInputStream/FileOutputStream` are for reading/writing bytes from/to files
- All the methods in `FileInputStream` and `FileOutputStream` are inherited from their superclasses



The `FileInputStream` class

<https://docs.oracle.com/javase/8/docs/api/java/io/FileInputStream.html>

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/FileInputStream.html>

- To construct a `FileInputStream` object, use the following constructors

```
public FileInputStream(String filename)
```

```
public FileInputStream(File file)
```

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

The `FileOutputStream` class

<https://docs.oracle.com/javase/8/docs/api/java/io/FileOutputStream.html>

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/FileOutputStream.html>

- To construct a `FileOutputStream` object, 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 will be created
- If the file already exists, the first two constructors will **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

Binary file I/O using `FileInputStream` and `FileOutputStream`

```
public class TestFileStream {
    public static void main(String[] args) throws IOException {
        try (
            // Create an output stream to the file
            FileOutputStream output = new FileOutputStream("temp.dat");
        ) {
            // Output values to the file
            for (int i = 1; i <= 10; i++)
                output.write(i);
        }

        try (
            // Create an input stream for the file
            FileInputStream input = new FileInputStream("temp.dat");
        ) {
            // Read values from the file
            int value;
            while ((value = input.read()) != -1) ←
                System.out.print(value + " ");
        }
    }
}
```

Use try-with-resources syntax
because classes implement
`AutoClosable` interface

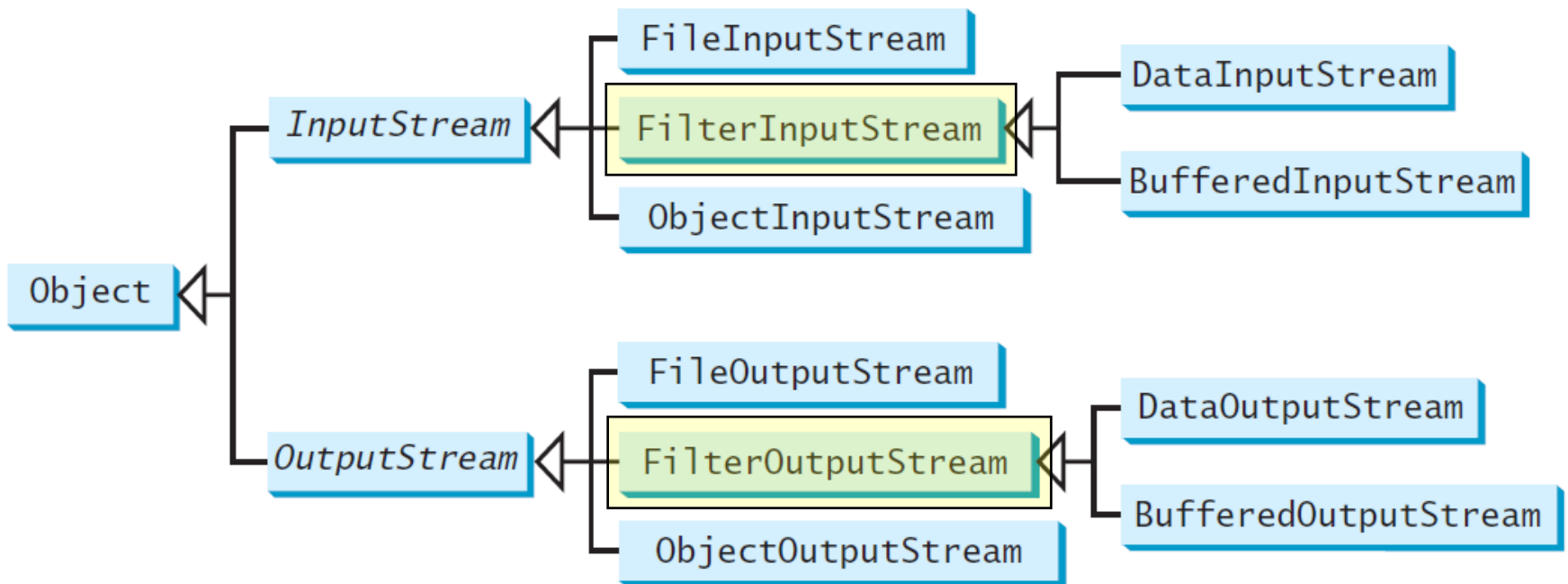
Check for end of file

Filter streams

- `FileInputStream` provides a `read` method that can only be used for reading bytes
 - If you want to read integers, doubles, or strings, you need a filter class to wrap the byte input stream
- *Filter streams* are streams that filter bytes for some purpose
 - Using a filter class enables you to read integers, doubles, and strings instead of bytes and characters

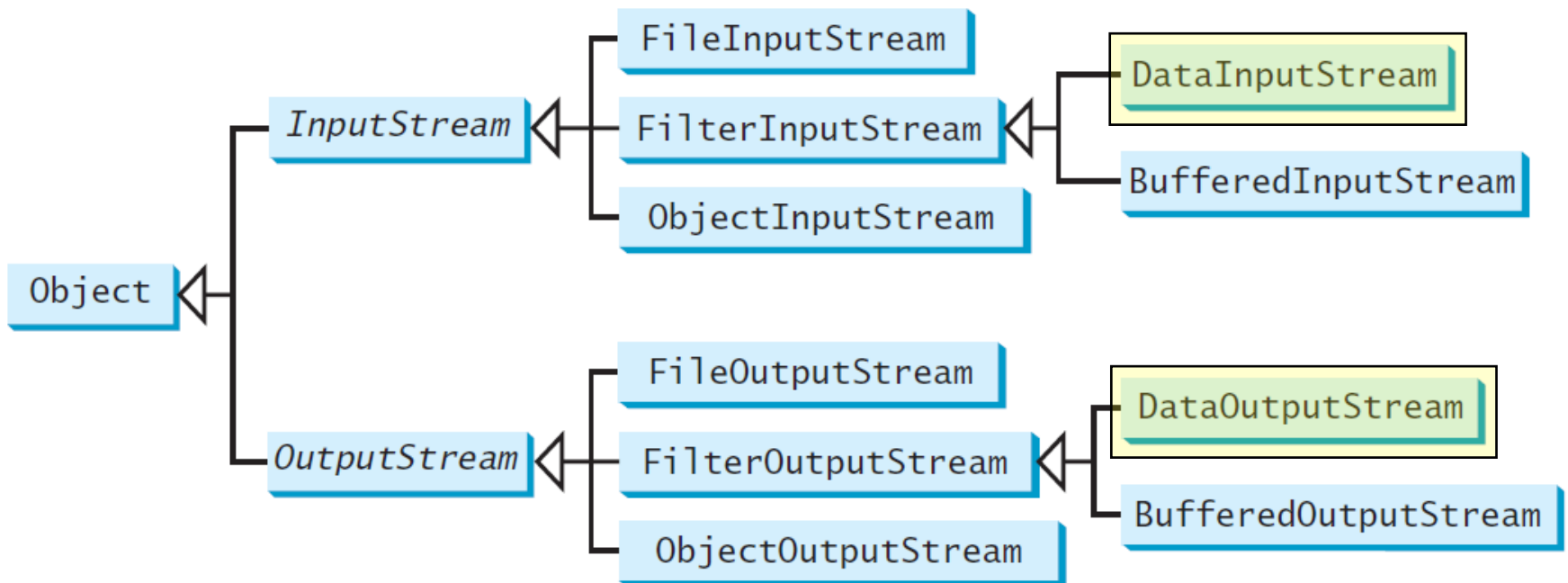
Binary filter I/O classes

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



Binary filter I/O classes

- When you need to process primitive numeric types, use `DataInputStream` and `DataOutputStream` to filter bytes

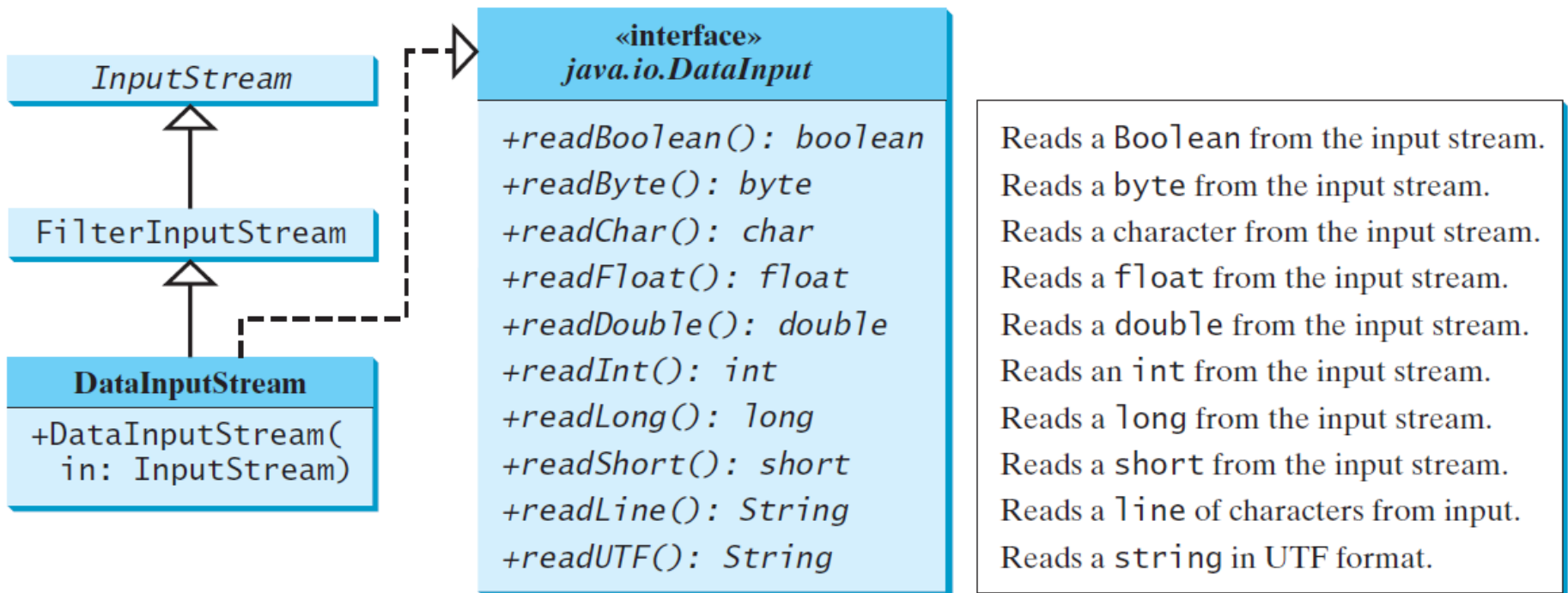


The DataInputStream class

<https://docs.oracle.com/javase/8/docs/api/java/io/DataInputStream.html>

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/DataInputStream.html>

- DataInputStream reads bytes from the stream and converts them into appropriate primitive type values or strings
- DataInputStream extends FilterInputStream and implements the DataInput interface

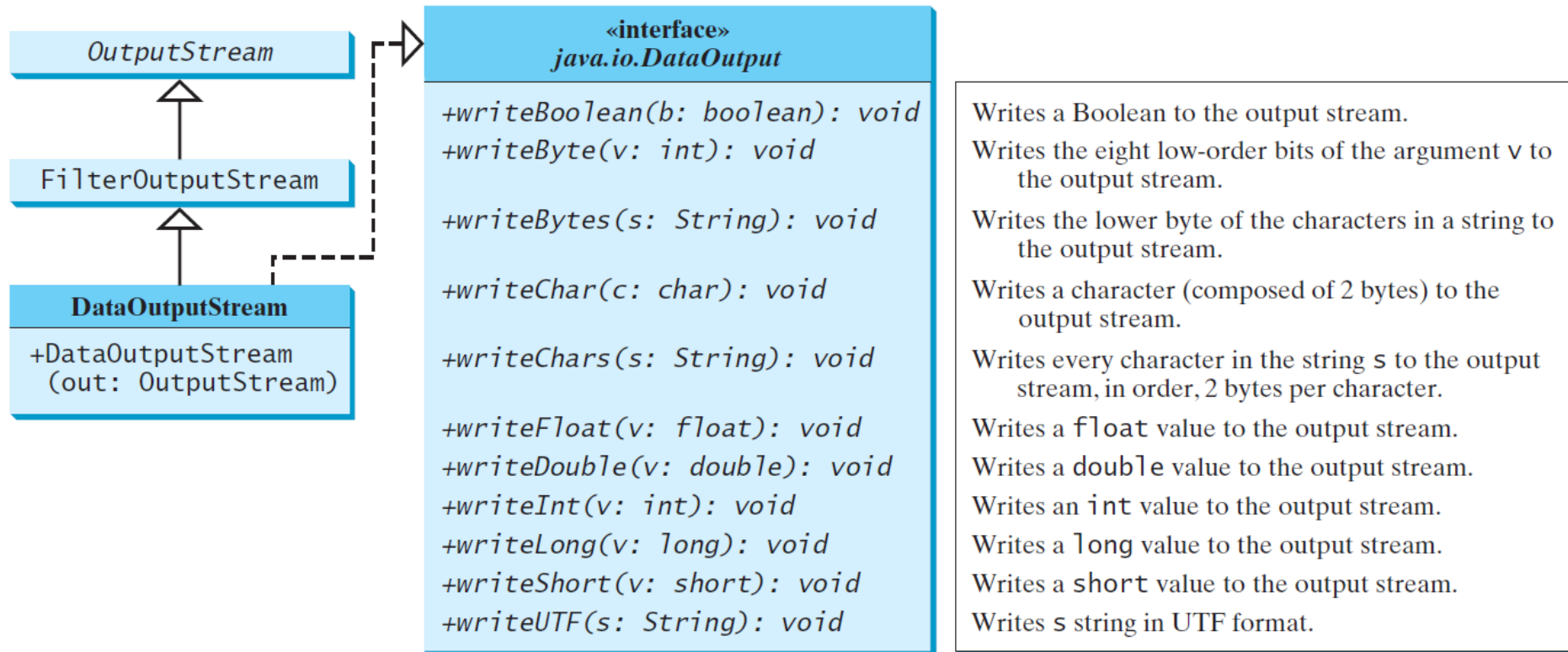


The DataOutputStream class

<https://docs.oracle.com/javase/8/docs/api/java/io/DataOutputStream.html>

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/DataOutputStream.html>

- DataOutputStream converts primitive type values or strings into bytes and output the bytes to the stream
- DataOutputStream extends FilterOutputStream and implements the DataOutput interface



Characters and strings in binary I/O

- Remember, 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
- Remember, an ASCII character consists of one byte, which is stored in the **lower byte** of a Unicode character
 - The `writeByte(int v)` method writes the **lowest byte** of integer `v` to the output (i.e., **the higher three bytes of the integer are discarded**)
 - The `writeBytes(String s)` method writes the **lower byte** of the Unicode of the characters in the string `s` to the output (i.e., **the higher byte of the Unicode of the characters are discarded**)

Characters and strings in binary I/O

- Unicode Transformation Format (UTF)
 - The `writeUTF(String s)` method writes the string `s` in UTF
 - UTF is coding scheme for efficiently compressing a string of Unicode characters

Binary file I/O using DataInputStream and DataOutputStream

```
public class TestDataStream {
    public static void main(String[] args) throws IOException {
        try ( // Create an output stream for file temp.dat
            DataOutputStream output =
                new DataOutputStream(new FileOutputStream("temp.dat"));
        ) {
            // Write student test scores to the file
            output.writeUTF("John");
            output.writeDouble(85.5);
            output.writeUTF("Jim");
            output.writeDouble(185.5);
            output.writeUTF("George");
            output.writeDouble(105.25);
        }

        try ( // Create an input stream for file temp.dat
            DataInputStream input =
                new DataInputStream(new FileInputStream("temp.dat"));
        ) {
            // Read student test scores from the file
            System.out.println(input.readUTF() + " " + input.readDouble());
            System.out.println(input.readUTF() + " " + input.readDouble());
            System.out.println(input.readUTF() + " " + input.readDouble());
        }
    }
}
```

You must read the data in the **same order** and **same format** in which they are stored

End of file (EOF)

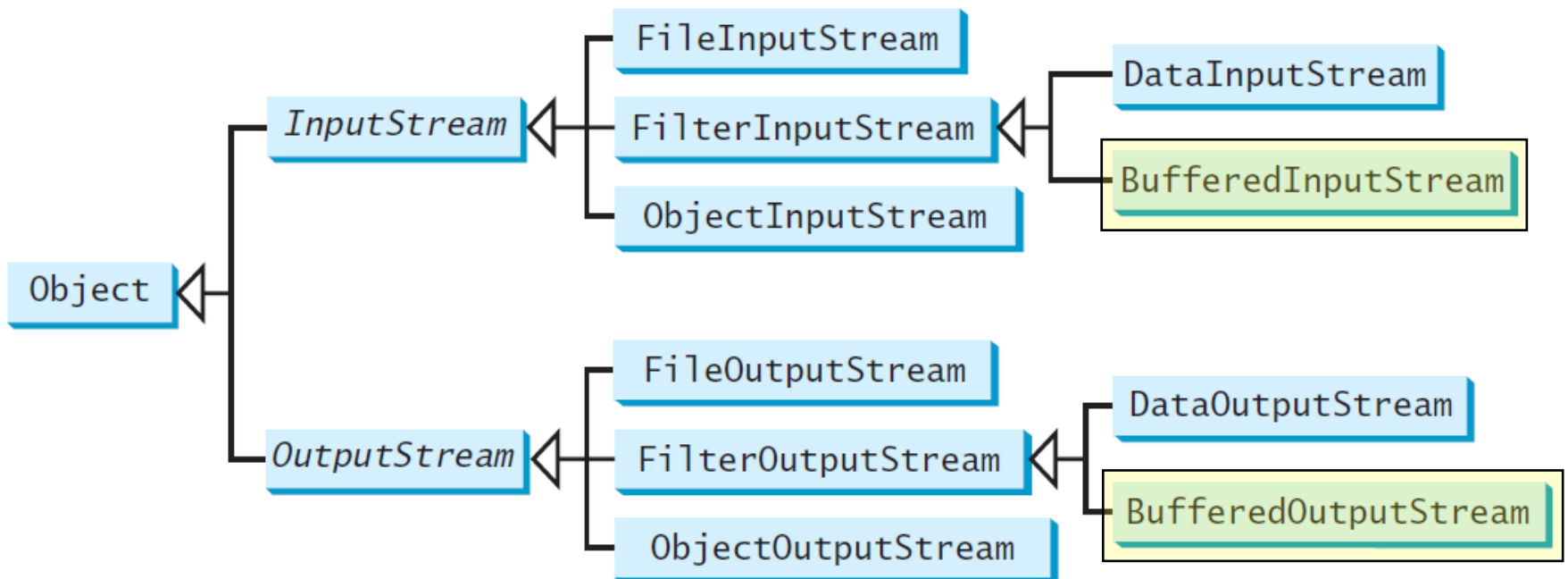
- If you keep reading data at the end of an `InputStream`, then an `EOFException` will occur

```
public class DetectEndOfFile {
    public static void main(String[] args) {
        try {
            try (DataInputStream input =
                new DataInputStream(new FileInputStream("test.dat"))) {
                while (true)
                    System.out.println(input.readDouble());
            }
        }
        catch (EOFException ex) {
            System.out.println("All data were read");
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

This is just an example to throw `EOFException`.
Use `input.available()` to check for EOF (if `input.available() == 0`, then it is EOF).

Binary filter I/O classes

- Use `BufferedInputStream` and `BufferedOutputStream` to speed up input and output by reading ahead and writing later
- All the methods in `BufferedInputStream` and `BufferedOutputStream` are inherited from their superclasses

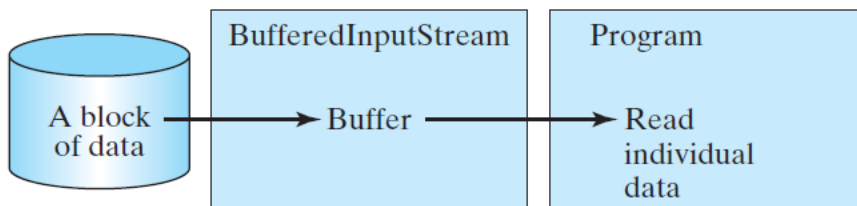


The BufferedInputStream and BufferedOutputStream classes

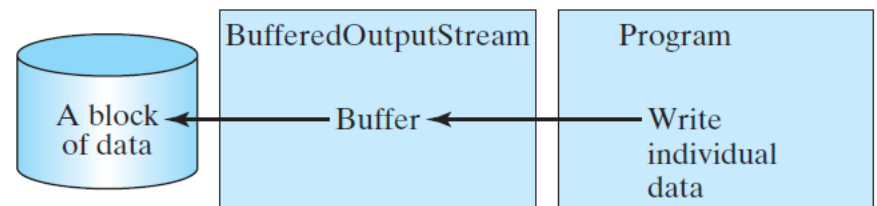
```
// Create a BufferedInputStream  
public BufferedInputStream(InputStream in)  
public BufferedInputStream(InputStream in, int bufferSize)
```

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

The default buffer size is 512 bytes



(a)



(b)

Example

```
public class Copy {
    public static void main(String[] args) throws IOException {
        // Check command-line parameter usage
        if (args.length != 2) {
            System.out.println(
                "Usage: java Copy sourceFile targetfile");
            System.exit(1);
        }

        // Check if source file exists
        File sourceFile = new File(args[0]);
        if (!sourceFile.exists()) {
            System.out.println("Source file " + args[0] + " does not exist");
            System.exit(2);
        }

        // Check if target file exists
        File targetFile = new File(args[1]);
        if (targetFile.exists()) {
            System.out.println("Target file " + args[1] + " already exists");
            System.exit(3);
        }
        ...
    }
}
```


Example

```
...
try (
    // Create an input stream
    BufferedInputStream input =
        new BufferedInputStream(new FileInputStream(sourceFile));

    // Create an output stream
    BufferedOutputStream output =
        new BufferedOutputStream(new FileOutputStream(targetFile));
) {
    // Continuously read a byte from input and write it to output
    int r;
    int numberOfBytesCopied = 0;
    while ((r = input.read()) != -1) { ← Check for end of file
        output.write((byte)r);
        numberOfBytesCopied++;
    }

    // Display the file size
    System.out.println(numberOfBytesCopied + " bytes copied");
}
}
```

Other binary file I/O

- Objects
 - `ObjectInputStream` and `ObjectOutputStream` can be used to read and write serializable objects
- Random access
 - `RandomAccessFile` allows data to be read from and written to any location (not necessarily sequentially) in the file

CSE 11 topics

Object-oriented programming

- Introduction to Java
- Numbers and mathematics
- Characters and strings
- Selections
- Methods
- Loops
- Recursion (simple)
- Arrays

Procedural programming

- Objects and classes
 - Object-oriented thinking
- Inheritance
- Polymorphism
- Abstract classes
- Interfaces
- Introduction to generics
- Exceptions
- Text file input/output
- Binary file input/output
- Assertions

Introduction to Java

- Java is:
 - a high-level programming language
 - Computer-specific details are abstracted
 - an object-oriented programming language
 - Based on classes
 - a strongly typed language
 - **Programmers must explicitly identify the type of every variable, method, and object**
 - a general-purpose programming language
 - Not specialized to a particular application domain
 - platform independent
 - Write a program once and run it on any computer

Numbers and mathematics

- Numerical data types (e.g., an integer)
- Numeric operations (e.g., addition)
- Mathematical functions (e.g., cosine)
- Reading numbers from the console

Characters and strings

- Character data type (i.e., `char`)
- Comparing and testing characters
- String data type (i.e., `String`)
- Simple string methods (e.g., number of characters in a string)
- Reading a character and string from the console

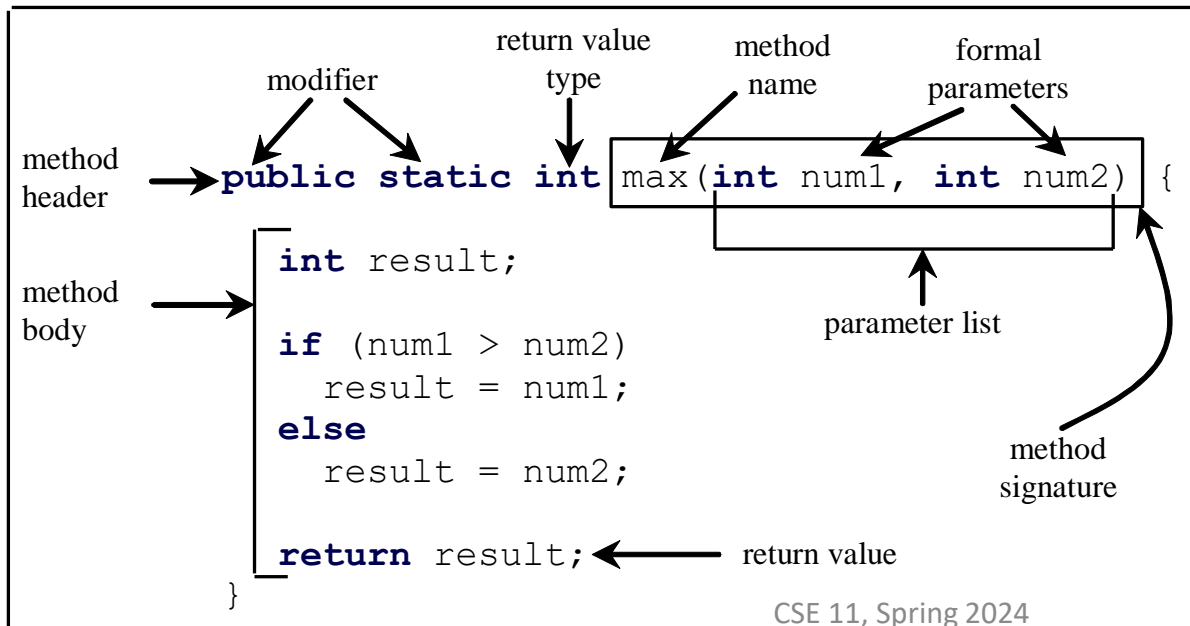
Selections

- Relational operators (e.g., less than, equal to)
- Logical operators (e.g., not, and, or)
- `if` statements
- `if-else` statements
- `switch` statements

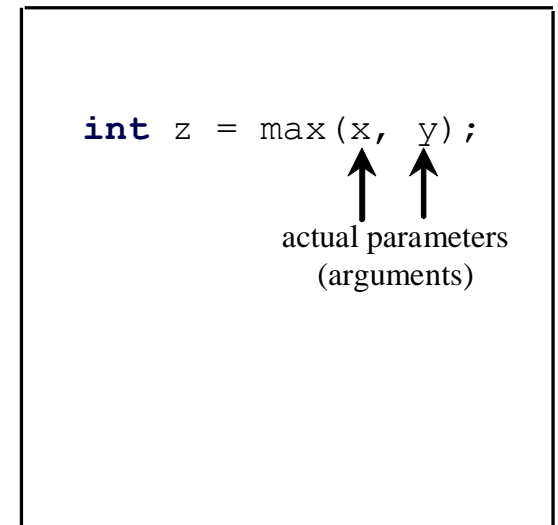
Methods

- A method is a collection of statements that are grouped together to perform an operation
- Write a method once and reuse it anywhere

Define a method



Invoke a method



Loops and recursion

- `while` loops
- `do-while` loops
- `for` loops
- Recursion is a technique that leads to elegant solutions to problems that are difficult to program using simple loops
 - A recursive method is one that invokes itself directly or indirectly

Arrays

- Array is a data structure that represents a collection of the same types of data

[0]	5.6
[1]	4.5
[2]	3.3
[3]	13.2
[4]	4.0
[5]	34.33
[6]	34.0
[7]	45.45
[8]	99.993
[9]	11123

← Element value

1-dimensional array

	[0]	[1]	[2]	[3]	[4]
[0]	0	0	0	0	0
[1]	0	0	0	0	0
[2]	0	7	0	0	0
[3]	0	0	0	0	0
[4]	0	0	0	0	0

2-dimensional array

Procedural programming vs object-oriented programming

- Procedural programming
 - Data and operations on data are separate
 - Requires passing data to methods
- Object-oriented programming
 - Data and operations on data are in an object
 - Organizes programs like the real world
 - All objects are associated with both attributes and activities
 - Using objects improves software reusability and makes programs easier to both develop and maintain

Objects and classes

- An object represents an entity in the real world that can be distinctly identified
 - For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects
 - An object has a unique identity, state, and behaviors
- Classes are constructs that define objects of the same type

Object-oriented thinking

- Classes provide more flexibility and modularity for building reusable software
- Class abstraction and encapsulation
 - Separate class implementation from the use of the class
 - The creator of the class provides a description of the class and let the user know how the class can be used
 - The user of the class does not need to know how the class is implemented
 - The detail of implementation is encapsulated and hidden from the user

Inheritance

- Inheritance enables you to define a general class (i.e., a *superclass*) and later extend it to more specialized classes (i.e., *subclasses*)
- A subclass inherits from a superclass
 - For example, both a circle and a rectangle are geometric objects
 - `GeometricObject` is a superclass
 - `Circle` is a subclass of `GeometricObject`
 - `Rectangle` is a subclass of `GeometricObject`
- Models **is-a** relationships
 - For example
 - `Circle` **is-a** `GeometricObject`
 - `Rectangle` **is-a** `GeometricObject`

Polymorphism

- A class defines a type
- A type defined by a subclass is called a *subtype*, and a type defined by its superclass is called a *supertype*
 - For example
 - Circle is a subtype of GeometricObject, and GeometricObject is a supertype for Circle
- *Polymorphism* means that a variable of a supertype can refer to a subtype object
 - Greek word meaning “many forms”

Abstract classes

- Inheritance enables you to define a general class (i.e., a *superclass*) and later extend it to more specialized classes (i.e., *subclasses*)
- Sometimes, a superclass is so general it cannot be used to create objects
 - Such a class is called an *abstract class*
- An abstract class cannot be used to create objects
- An **abstract** class can contain abstract methods that are implemented in **concrete** subclasses
- Just like nonabstract classes, models **is-a** relationships
 - For example
 - Circle **is-a** GeometricObject
 - Rectangle **is-a** GeometricObject

Interfaces

- An interface is a class-like construct that contains **only** constants and abstract methods
 - In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify common behavior for objects
 - For example, you can specify that the objects are comparable and/or cloneable using appropriate interfaces
- Interfaces model **is-kind-of** relationships
 - For example
 - Fruit **is-kind-of** Edible
 - Fish **is-kind-of** Edible

Methods and data fields visibility

Modifiers on Members in a Class	Accessed from the Same Class	Accessed from the Same Package	Accessed from a Subclass in a Different Package	Accessed from a Different Package
Public	✓	✓	✓	✓
Protected	✓	✓	✓	
Default (no modifier)	✓	✓		
Private	✓			

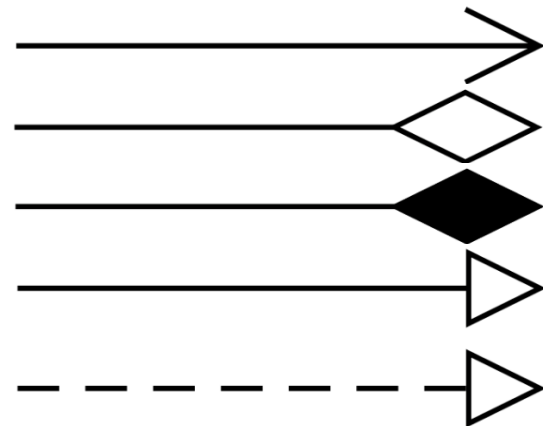
Unified Modeling Language (UML)

+ public

protected

- private

- Static variables and methods are underlined
- Abstract class names and methods are *italicized*
- Interface names and methods are *italicized*
- Open or no arrow is association
- Hollow diamond is aggregation
- Filled diamond is composition
- Hollow triangle is inheritance
- Dashed line with hollow triangle is implementation of interface



Additional topics

- Introduction to generics
- Exceptions
- Text file input/output (I/O)
- Binary file input/output (I/O)
- Assertions

Introduction to generics

- Generics let you parameterize types
 - You can define a method or class with generic types, which are replaced with concrete types

Exceptions

- Exceptions are errors caused by your program and external circumstances
 - These errors can be caught and handled by your program
- Exception handling separates error-handling code from normal programming tasks
 - Makes programs easier to read and to modify
- The **try** block contains the code that is executed in **normal** circumstances
- The **catch** block contains the code that is executed in **exceptional** circumstances
- A method should **throw** an exception if the error needs to be handled by its caller

Text file input/output (I/O)

- 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

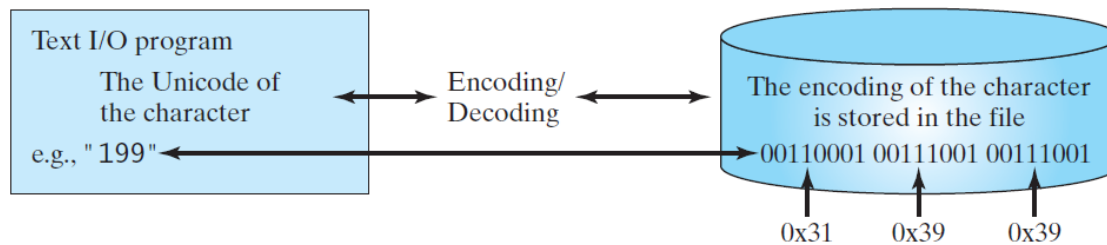
File

Scanner

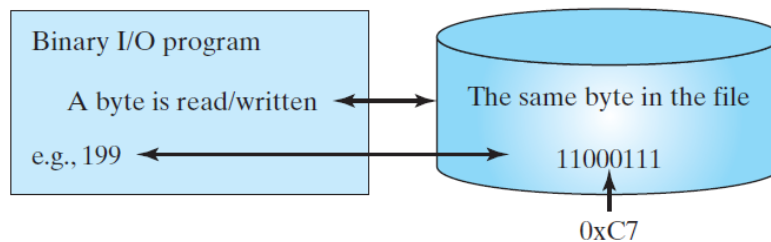
PrintWriter

Binary file input/output (I/O)

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



(a)



(b)

Assertions

- An assertion is a Java statement that enables you to assert an assumption about your program
- An assertion contains a Boolean expression that should be true during program execution
- Assertions can be used to assure program correctness and avoid logic errors

CSE 11 topics

Object-oriented programming

- Introduction to Java
- Numbers and mathematics
- Characters and strings
- Selections
- Methods
- Loops
- Recursion (simple)
- Arrays

Procedural programming

- Objects and classes
 - Object-oriented thinking
- Inheritance
- Polymorphism
- Abstract classes
- Interfaces
- Introduction to generics
- Exceptions
- Text file input/output
- Binary file input/output
- Assertions