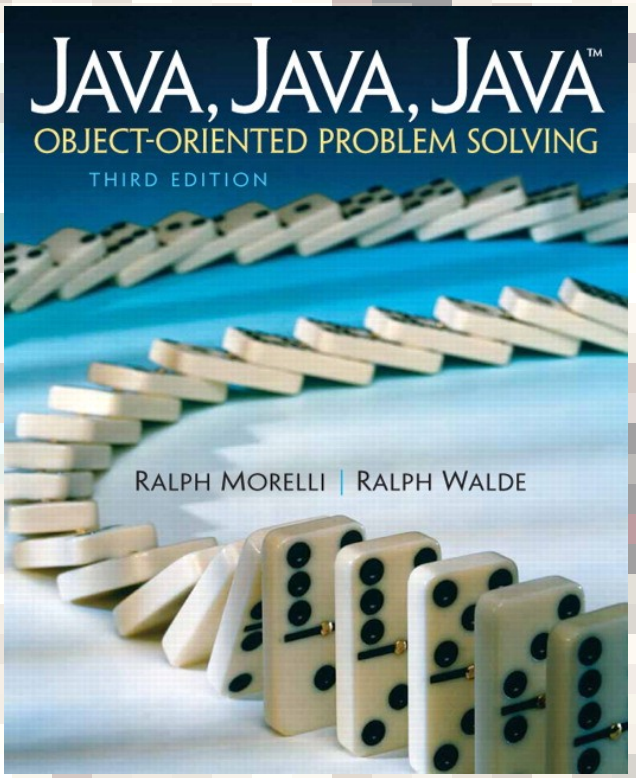


presentation slides for

JAVA, JAVA, JAVA

Object-Oriented Problem Solving

Third Edition

Ralph Morelli | Ralph Walde
Trinity College
Hartford, CT

published by Prentice Hall

Java, Java, Java

Object Oriented Problem Solving

Lecture 04: Interfaces and Files, Streams, and Input/Output Techniques

Objectives

- Review of Interface and its usage
- Be able to read and write text and binary files.
- Understand the use of InputStream and OutputStream
- Be able to design methods for performing input and output.
- Know how to use the File, JFileChooser

Outline

- Part 1: Interface
- Part 2: File, Stream, and I/O Revised
  - Streams and Files
  - Case Study: Reading and Writing Files
  - The File Class
  - Reading and Writing Binary Files
  - Reading and Writing Objects
  - From the Java Library: JFileChooser
  - Using the File Data in Programs

Interface – Common behaviors of types

- Consider the task of writing classes to represent 2D shapes such as Circle, Rectangle, and Triangle.
- There are certain operations that are common to all shapes: calculating area and perimeter area
- By being a Shape, you promise that you can compute those attributes, but each shape computes them differently.

Interface – Behavior Contract

- Analogous to the idea of roles or certifications in real life:
  - "I'm certified as a CPA accountant. The certification assures you that I know how to do taxes, perform audits."
- Compare to:
  - "I'm certified as a Shape. That means you can be sure that I know how to compute my area and perimeter."

Calculating area and perimeter of shapes

- Rectangle (based on width w and height h)
  - Area = w * h
  - Perimeter = 2 (w + h)
- Circle (based on radius r)
  - Area = πr^2
  - Perimeter = $2\pi r$
- Triangle (based on three sides a, b, c)
  - Area = $\sqrt{s(s-a)(s-b)(s-c)}$ where $s = \frac{1}{2}(a+b+c)$ according to Heron's formula
  - Perimeter = a + b+ c

Interfaces

- Interface: A list of methods that a class promises to implement.
  - Inheritance gives you an is-a relationship and code-sharing.
    - An Executive object can be treated as a StaffMember,
    - Executive inherits StaffMember's code.
  - Interfaces give you an is-a relationship without code sharing.
    - Only abstract methods in the interface
    - Object can-act-as any interface it implements
    - A Rectangle object can be treated as a Shape as long as it implements the interface.

Declare Interface in Java

- Interface declaration syntax:

```
public interface <name> {
    public <type> <name>( <parameter list> );
    public <type> <name>( <parameter list> );
}
```

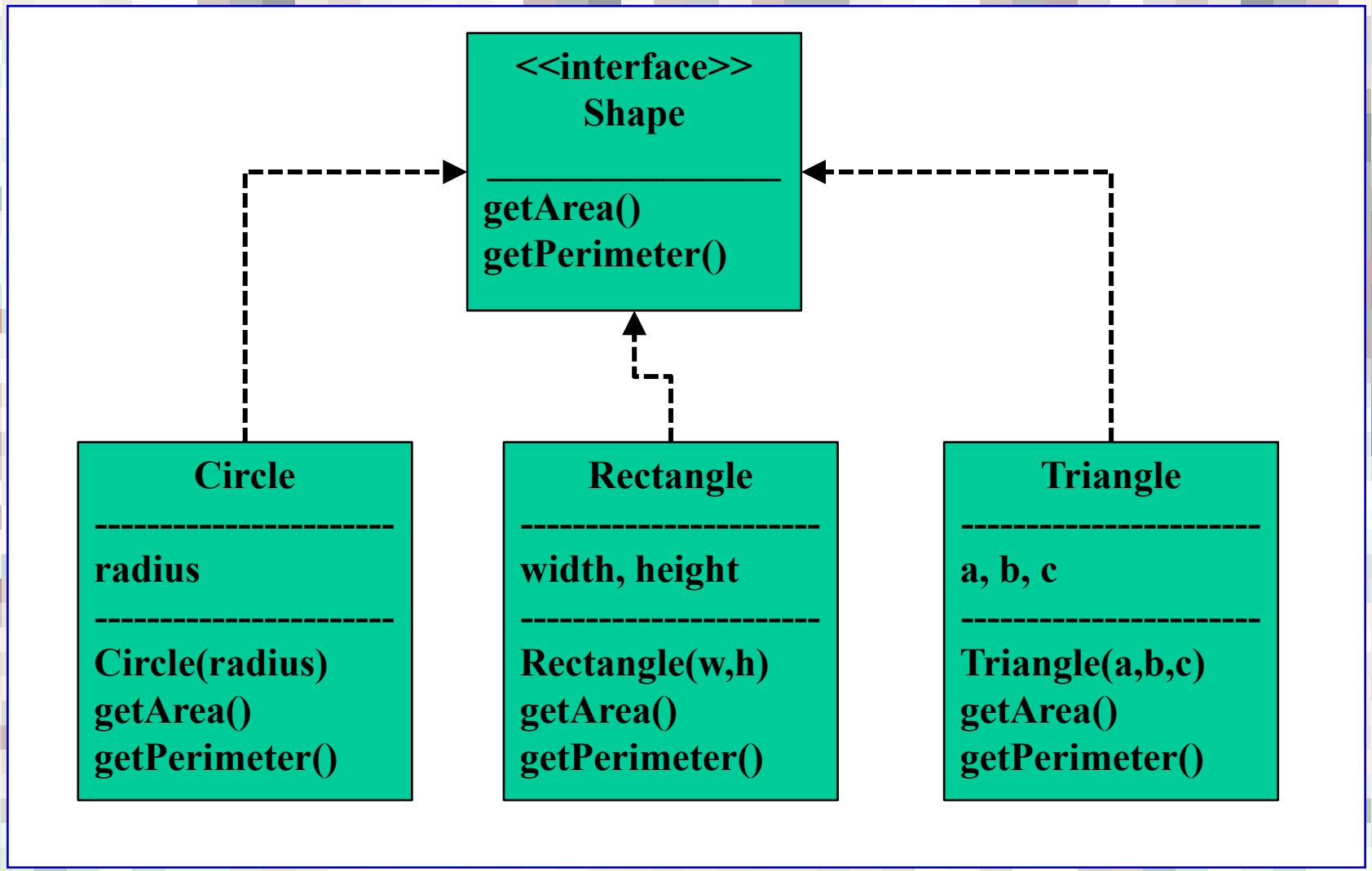
- **Note**:
  - All methods are public!
  - All methods are abstract, and no need to put the abstract keyword
  - No methods have the implementation

Implementing Interfaces

- Any Java classes can implement one or more interfaces
- A class implements an interface, it needs to define the implementations of all abstract methods in that interface
- Syntax

```
public class <name> implements <interfaces> {
        …
}
```

Interface Example



Interfaces and Polymorphism

- A class implements an interface can take the advantage of polymorphism => Interface is also a type!
- Example:

```
private static void print(Shape s) {
    System.out.println("Shape: " + s + " area: "
      + s.getArea() + " perimeter: " + s.getPerimeter());
}
```

- Any object that implements the interface may be passed as the parameter to the above method, i.e.
  - print(new Circle(3.0))
  - print(new Rectangle(2,3))

Interfaces and Polymorphism

- We can create an array of an interface type, and store any object implementing that interface as an element.

```
Circle circ = new Circle(12.0);
Rectangle rect = new Rectangle(4, 7);
Triangle tri = new Triangle(5, 12, 13);
Shape[] shapes = { circ, tri, rect };
for (int i = 0; i < shapes.length; i++) {
    print(shapes[i]);
}
```

- Each element of the array executes the appropriate behavior for its object when it is passed to the print method, or when getArea() or getPerimeter() is called on it.

When to use Interfaces

- Think of an interface as an abstract base class with all abstract methods
- Interfaces are used to define a contract for how you interact with an object, independent of the underlying implementation
- Separation behavior (interface) from the implementation
- Use interfaces to get around the Java limitation of single inheritance – a class can only extend another class but can implement multiple interfaces
- Interface can be inherited – extends from another interfaces => add more abstract methods - behaviors

Commonly used Java interfaces

- The Java class library contains classes and interfaces
- Comparable – allows us to order the elements of an arbitrary class
- Serializable (in java.io) – for saving objects to a file.
- List, Set, Map, Iterator (in java.util) – describe data structures for storing collections of objects

Comparable interfaces

- A class can implement the Comparable interface to define an ordering for its objects.

```
public interface Comparable<E> {
  public int compareTo(E other);
}

public class Employee implements Comparable<Employee> { … }
```

- A call of a.compareTo(b) should return:
  - a value < 0 if a comes "before" b in the ordering,
  - a value > 0 if a comes "after" b in the ordering,
  - or 0 if a and b are considered "equal" in the ordering

Comparable interfaces (cont.)

- If you implement Comparable, you can sort arbitrary objects using the method Arrays.sort()
- The class type of those objects needs to implement the method compareTo().
- Delegation trick - If your object's attributes are comparable (such as strings), you can use their compareTo:

```
// sort by employee name
public int compareTo(Employee other) {
    return name.compareTo(other.getName());
}
```


ArrayList and List interface

- The ArrayList declaration:

  public class ArrayList<E> extends AbstractList<E> implements

  List<E>, RandomAccess, Cloneable, Serializable

- The List<E> interface defines the following methods: get(index), indexOf(object), remove(index), and set(index, object)
- The declaration of the List interface:

  public interface List<E> extends Collection<E>

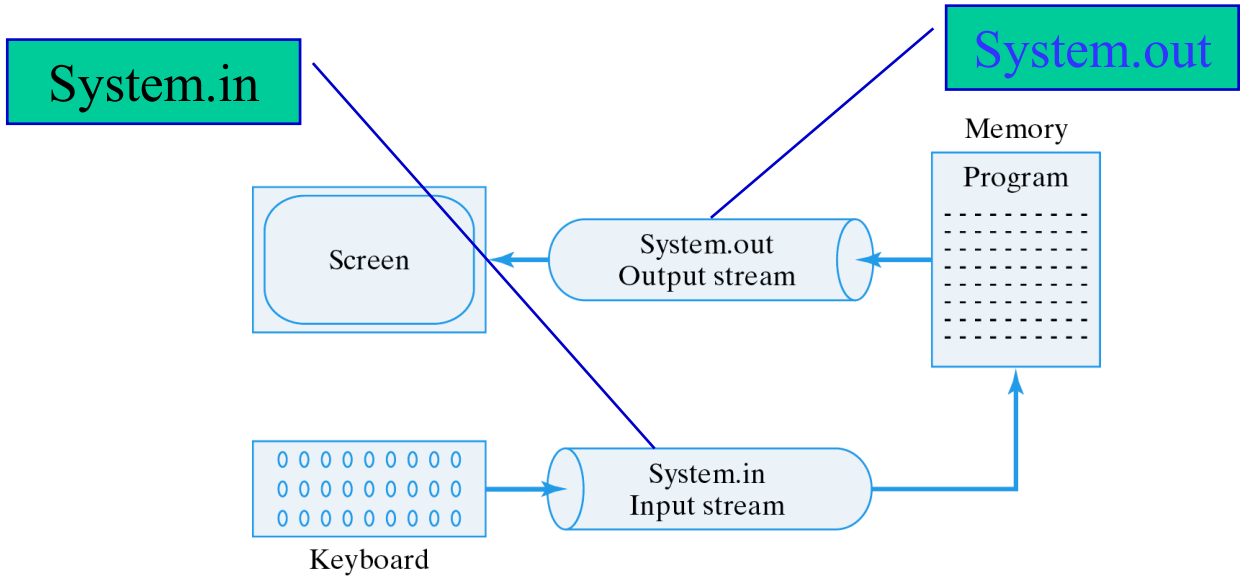
- It has methods that any collection of elements should have such as add, clear(), contains, isEmpty(), remove, size()

Introduction

- *Input* refers to *reading* data from some external source into a running program.
- *Output* refers to *writing* data from a running program to some external destination.
- A *file* is a collection of data stored on a disk or CD or some other storage medium.
- Files and their data persist beyond the duration of the program.

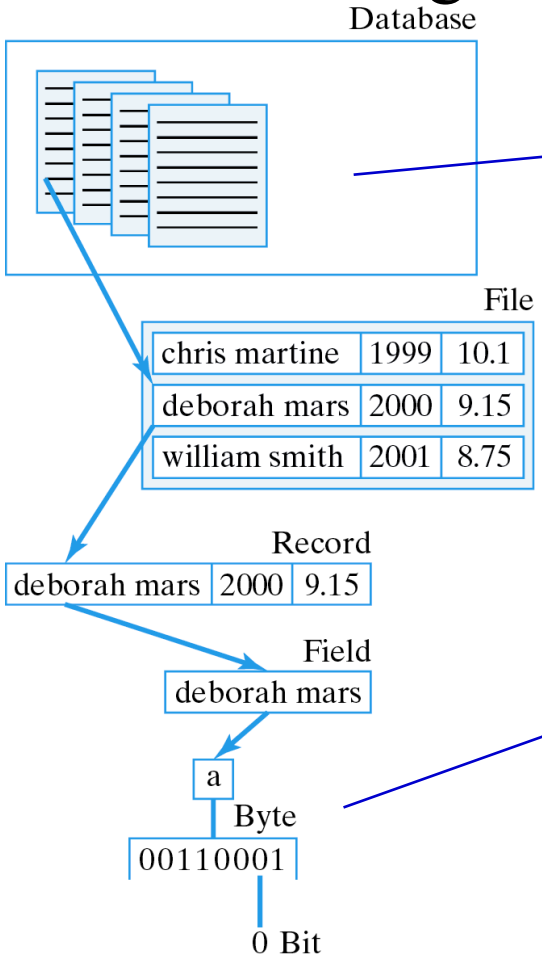
Streams and Files

- A *stream* is an object that delivers information to another object or from another object.
- All I/O in Java is based on streams.



The Data Hierarchy

- Data can be arranged in a hierarchy.



A *database* is a collection of *files*.

A *byte* is a collection of *bits*.

Binary Files and Text Files

- A *binary file* is processed as a sequence of bytes, whereas a *text file* is processed as a sequence of characters. Both types store data as a sequence of bits and bytes (0's and 1's).
- Text files are *portable* because they are based on the *ASCII code*.
- Generally, binary files are not portable because they use different representations of binary data.
- But Java binary files are *platform independent* because Java defines the sizes of binary data.

Which Stream to Use?

- Binary I/O: For binary I/O we use subclasses of InputStream and OutputStream.
- Text I/O: Subclasses of Reader and Writer are normally used for text I/O.

Buffering

- A *buffer* is a relatively large region of memory used to temporarily store data during I/O.
- BufferedInputStream and BufferedOutputStream are designed for this purpose.
- Buffering improves efficiency.
- The StringReader and StringWriter classes provide methods for treating Strings and StringBuffers as I/O streams.

Writing to a Text File

- Text file format: a sequence of characters divided into 0 or more lines and ending with a special *end-of-file* character.

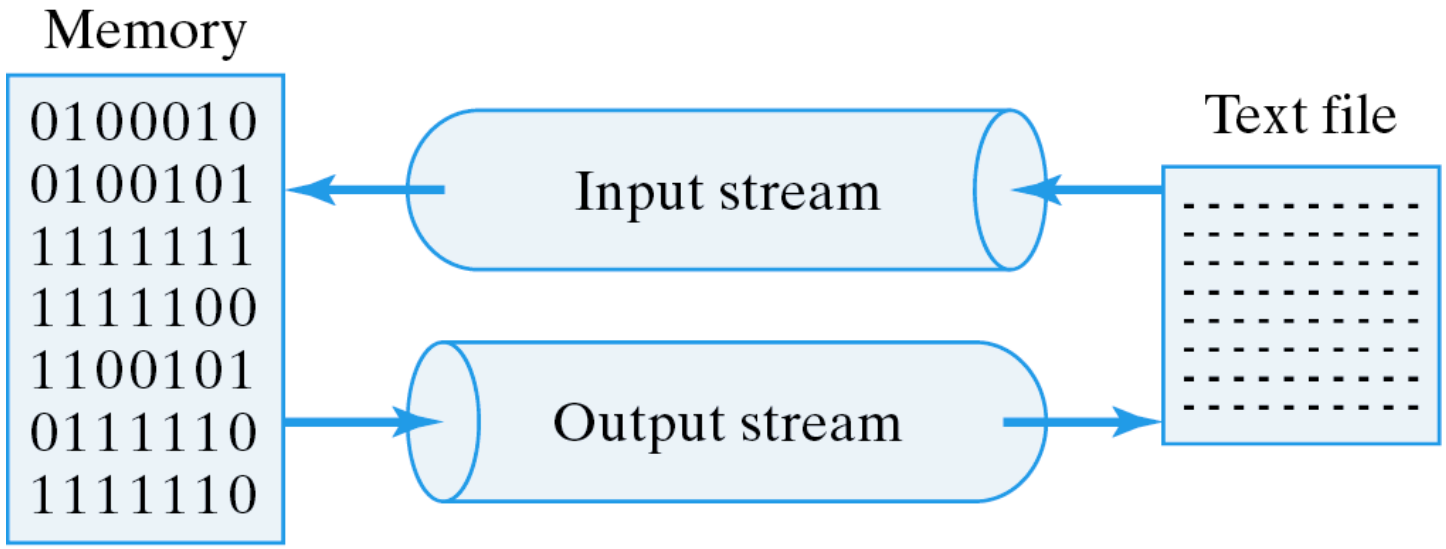
```
one\ntwo\nthree\nfour\eof
```

End-of-line (\n) and end-of-file (\eof) characters.

- Basic algorithm:
  - Connect an output stream to the file.
  - Write text into the stream, possibly with a loop.
  - Close the stream.

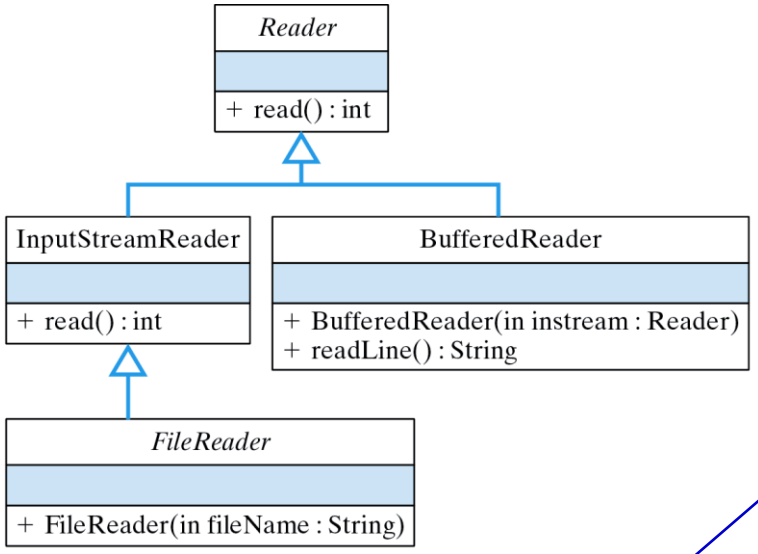
Reading from a Text File

- Basic algorithm:
  - Connect an input stream to the file.
  - Read the text data using a loop.
  - Close the stream.



Choosing methods and streams

- We need the FileReader(filename) constructor and the BufferedReader readLine(String) method



Connecting the streams.

```
BufferedReader inStream
      = new BufferedReader(new FileReader(fileName));
```

Self-defined readTextFile() Method

```
private void readTextFile(JTextArea display, String fileName) {
     try {
         BufferedReader inStream                  // Create and open the stream
            = new BufferedReader (new FileReader(fileName));
         String line = inStream.readLine();  // Read one line
         while (line != null) {                   // While more text
             display.append(line + "\n");      // Display a line
             line = inStream.readLine();         // Read next line
         }
         inStream.close();                        // Close the stream
     } catch (FileNotFoundException e) {
         display.setText("IOERROR: File NOT Found: " + fileName + "\n");
         e.printStackTrace();
     } catch ( IOException e ) {
         display.setText("IOERROR: " + e.getMessage() + "\n");
         e.printStackTrace();
     }
} // readTextFile()
```

readLine() returns null at *end-of-file*

Exception handling.

The Read Loop

- Read loops are designed to work on empty files.

```
try to read one line of data and store it in line  // Loop initializer
  while (line is not null) { // Loop entry condition
    process the data
    try to read one line of data and store it in line // Loop updater
  }
```

A while loop iterates 0 or more times.

Code Reuse: Designing Text File Input

- We can also read one character at a time:

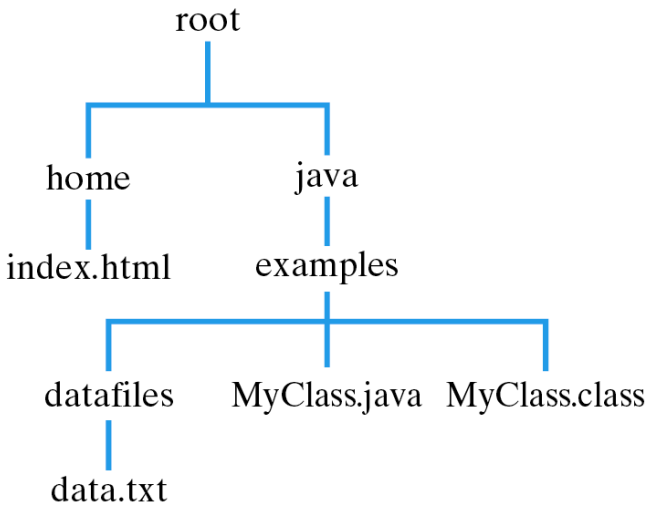
```
int ch = inStream.read();    // Initializer: try to read the next character
while (ch != -1) {           // Loop-entry-condition: while more characters
     display.append((char)ch + ""); // Append the character
     ch = inStream.read();          // Updater: try to read next character
}
```

- Basic file reading loop:

```
try to read data into a variable              // Loop initializer
    while ( read was successful ) {          // Loop entry condition
        process the data
        try to read data into a variable     // Loop updater
    }
```

Files and Paths

- A *path* is a description of a file's location in its hierarchy:



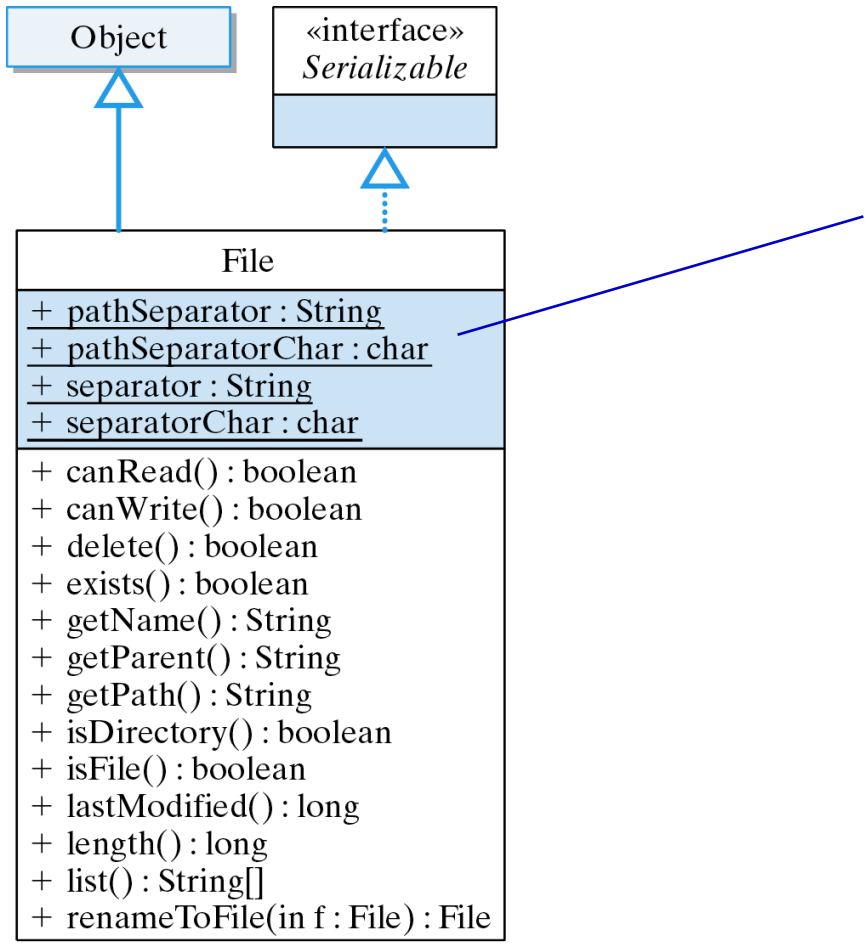
- *Absolute* path name:

  /root/java/example/MyClass.java

- *Relative* path name:

  MyClass.java

The File Class



Platform independence:
Unix: /
Windows: \

The isReadableFile() Method

```
private boolean isReadableFile(String fileName) {
     try {
         File file = new File(fileName);
         if (!file.exists())
             throw (new FileNotFoundException("No such File:" + fileName));
         if (!file.canRead())
             throw (new IOException("File not readable: " + fileName));
         return true;
     } catch (FileNotFoundException e) {
         System.out.println("IOERROR: File NOT Found: " + fileName + "\n");
          return false;
     } catch (IOException e) {
         System.out.println("IOERROR: " + e.getMessage() + "\n");
         return false;
     }
} // isReadableFile
```

Create a file object from the file name.

Check existence and readability.

The isWriteableFile() Method

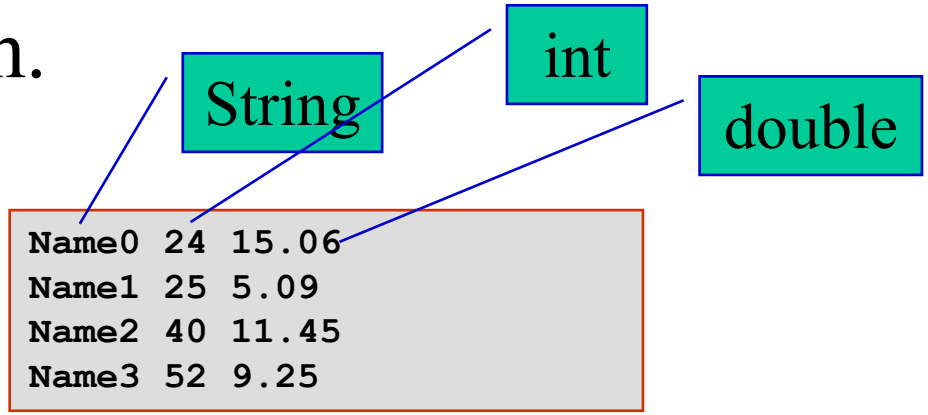
```
private boolean isWriteableFile(String fileName) {
    try {
        File file = new File (fileName);
        if (fileName.length() == 0)
            throw (new IOException("Invalid file name: " + fileName));
        if (file.exists() && !file.canWrite())
         throw (new IOException("IOERROR: File not writeable: " + fileName));
        return true;
    } catch (IOException e) {
        display.setText("IOERROR: " + e.getMessage() + "\n");
        return false;
    }
} // isWriteableFile()
```

Create a file object from the file name.

Check writeability.

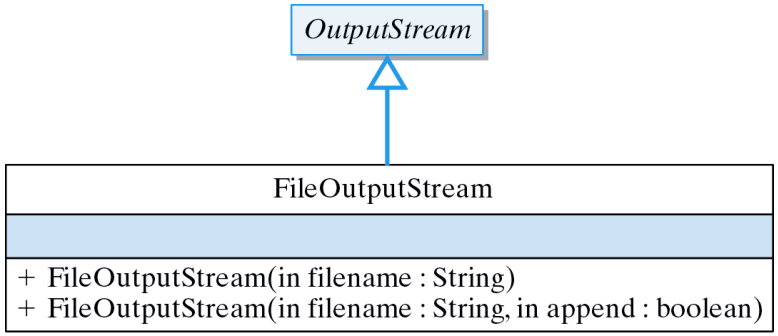
Reading and Writing Binary Files

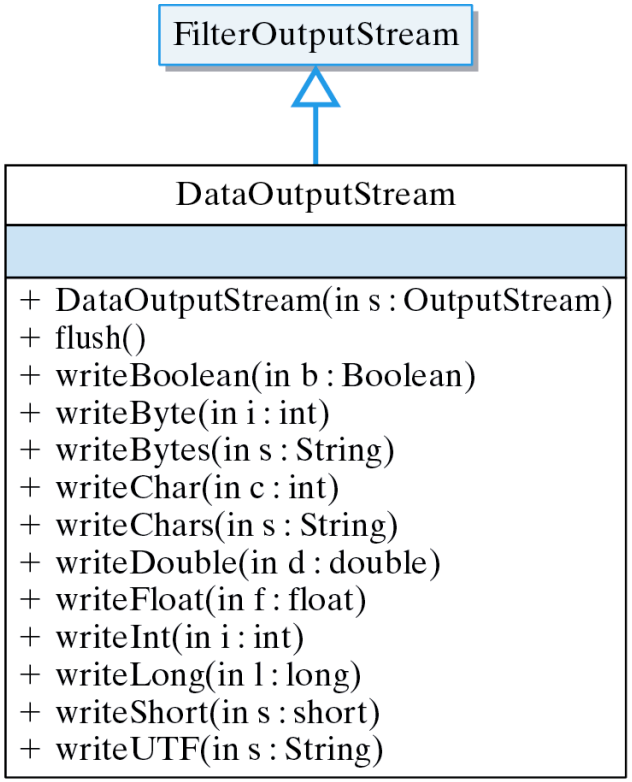
- Binary files have NO end-of-file character.
- Basic algorithm:
  - Connect a stream to the file.
  - Read or write the data, possibly using a loop.
  - Close the stream.
- Sample Data:



Output Method Design

- What streams and methods to use?
- DataOutputStream contains the right methods.
- FileOutputStream has the right constructor.





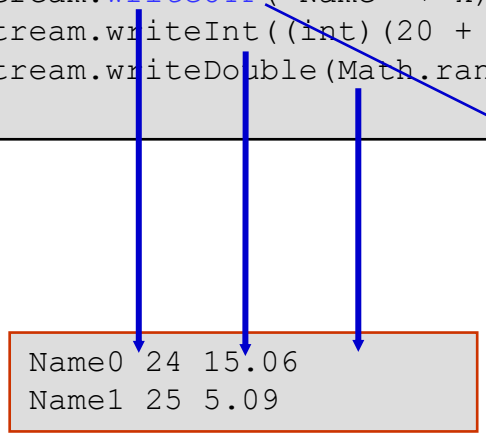
Writing Binary Data

- Connecting the streams to the file:

```
DataOutputStream outStream
          = new DataOutputStream(new FileOutputStream (fileName));
```

- Writing data to the file:

```
for (int k = 0; k < 5; k++) {                    // Output 5 data records
  outStream.writeUTF("Name" + k);                                  // Name
  outStream.writeInt((int)(20 + Math.random() * 25)); // Random age
  outStream.writeDouble(Math.random() * 500);  // Random payrate
}
```



Unicode Transformation Format (UTF) is a coding scheme for Java's Unicode character set.

Self-defined writeRecords() Method

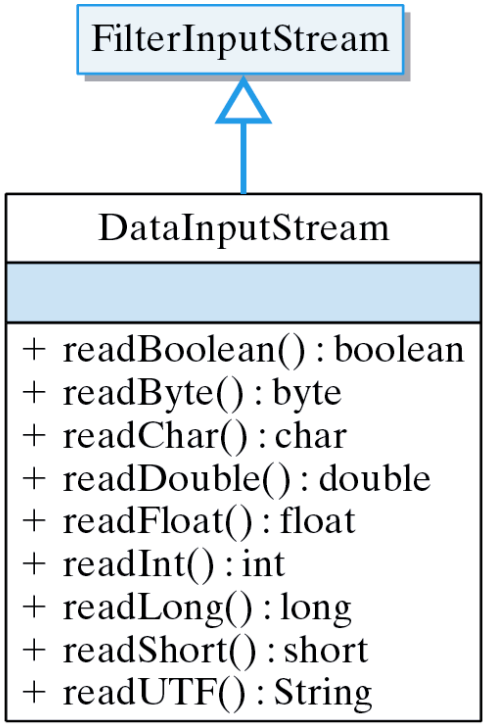
```
private void writeRecords( String fileName )  {
    try {
        DataOutputStream outStream                  // Open stream
              = new DataOutputStream(new FileOutputStream(fileName));
        for (int k = 0; k < 5; k++) {                // Output 5 data records
             String name = "Name" + k;
             outStream.writeUTF("Name" + k);                          // Name
             outStream.writeInt((int)(20 + Math.random() * 25)); // Age
             outStream.writeDouble(5.00 + Math.random() * 10);    // Payrate
         } // for
         outStream.close();                                  // Close the stream
    } catch (IOException e) {
         display.setText("IOERROR: " + e.getMessage() + "\n");
    }
} // writeRecords()
```

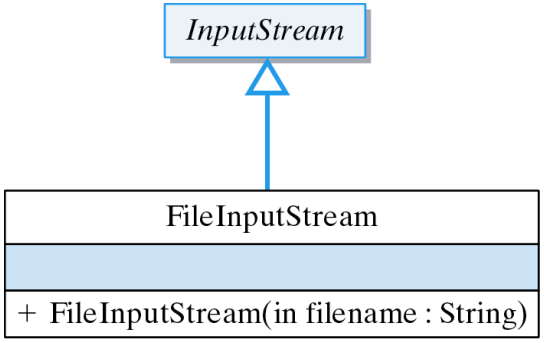
Write 5 records.

Handle exception.

Input Method Design

- What streams and methods to use?
- DataInputStream contains the right methods.
- FileInputStream has the right constructor.





The Read Loop

- Binary files have no end-of-file marker.

```
try {
    while (true) {                                  // Infinite loop
        String name = inStream.readUTF();           // Read a record
        int age = inStream.readInt();
        double pay = inStream.readDouble();
        display.append(name + "   " + age + "   " + pay + "\n");
    } // while
} catch (EOFException e) {}                         // Until EOF exception
```

Data input routine matches data output routine.

End of file is signaled by EOFException.

The readRecords() Method

```
private void readRecords(String fileName) {
    try {
        DataInputStream inStream
           = new DataInputStream(new FileInputStream(fileName)); // Open stream
        display.setText("Name   Age Pay\n");
        try {
            while (true) {                                    // Infinite loop
                String name = inStream.readUTF();           // Read a record
                int age = inStream.readInt();
                double pay = inStream.readDouble();
                display.append(name + "   " + age + "    " + pay + "\n");
            } // while
        } catch (EOFException e) {
        } finally {
            inStream.close();
        }
    } catch (FileNotFoundException e) {
        display.setText("IOERROR: File NOT Found: " + fileName + "\n");
    } catch (IOException e) {
        display.setText("IOERROR: " + e.getMessage() + "\n");
    }
} // readRecords()
```

Connect streams.

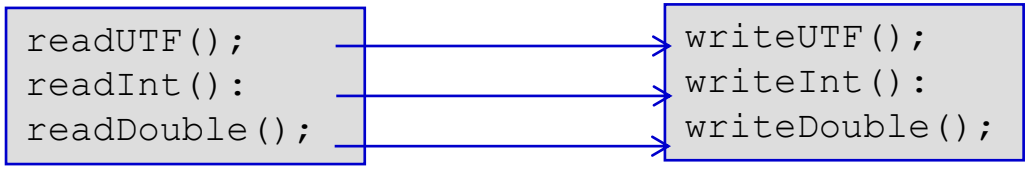
EOF Nested try block.

Note finally.

Catch IOExceptions.

Abstracting Data from Files

- Binary read routine must match write routine:



- A binary file is a sequence of 0's and 1's:

```
01010011001100100101010011001100000101001100100101010011001100
01100101101010011001100
```

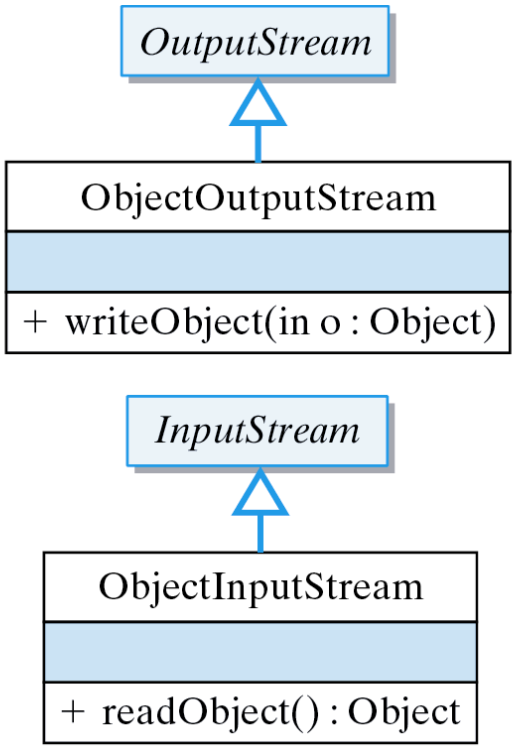
- The above sequence is interpretable as two 32-bit ints or one 64-bit double or eight 8-bit ASCII characters.
- **Effective Design: Data Abstraction.** Data are raw. The program determines type.

Reading and Writing Objects

- Object *serialization* is the process of writing an object as a series of bytes.
- *Deserialization* is the opposite (input) process.

Object I/O Classes

- ObjectOutputStream is used for object output.
- ObjectInputStream is used for object input.



The Student Class

```
import java.io.*;
public class Student implements Serializable {
    private String name;
    private int year;
    private double gpr;

    public Student() {}

    public Student (String nameIn, int yr, double gprIn) {
        name = nameIn;
        year = yr;
        gpr = gprIn;
    }

    public String toString() {
        return name + "\t" + year + "\t" + gpr;
    }
} // Student
```

An object can contain other objects.

The Student Serialization Methods

```
public void writeToFile(FileOutputStream outStream)
                                                  throws  IOException
{
    ObjectOutputStream ooStream = new
              ObjectOutputStream(outStream);
    ooStream.writeObject(this);
    ooStream.flush();
} // writeToFile()

public void readFromFile(FileInputStream inStream)
                         throws IOException,  ClassNotFoundException
{
    ObjectInputStream oiStream = new ObjectInputStream(inStream);
    Student s = (Student)oiStream.readObject();
    this.name = s.name;
    this.year = s.year;
    this.gpr = s.gpr;
} // readFromFile()
```

Recursively writes the object to the stream.

Recursively reads the object from the stream.

The writeRecords() Method

```
private void writeRecords(String fileName) {
    try {
        FileOutputStream outStream = new FileOutputStream(fileName);
        for (int k = 0; k < 5; k++) {            // Generate 5 random objects
            String name = "name" + k;
            int year = (int)(2000 + Math.random() * 4);
            double gpr = Math.random() * 12;
            Student student = new Student(name, year, gpr); // Object
            display.append("Output: " + student.toString() +  "\n");
            student.writeToFile(outStream) ;      //  and write it to file
         } //for
         outStream.close();
     } catch (IOException e) {
         display.append("IOERROR: " + e.getMessage() + "\n");
     }
} // writeRecords()
```

Serialization

Note: writeToFile() is defined in the last page.

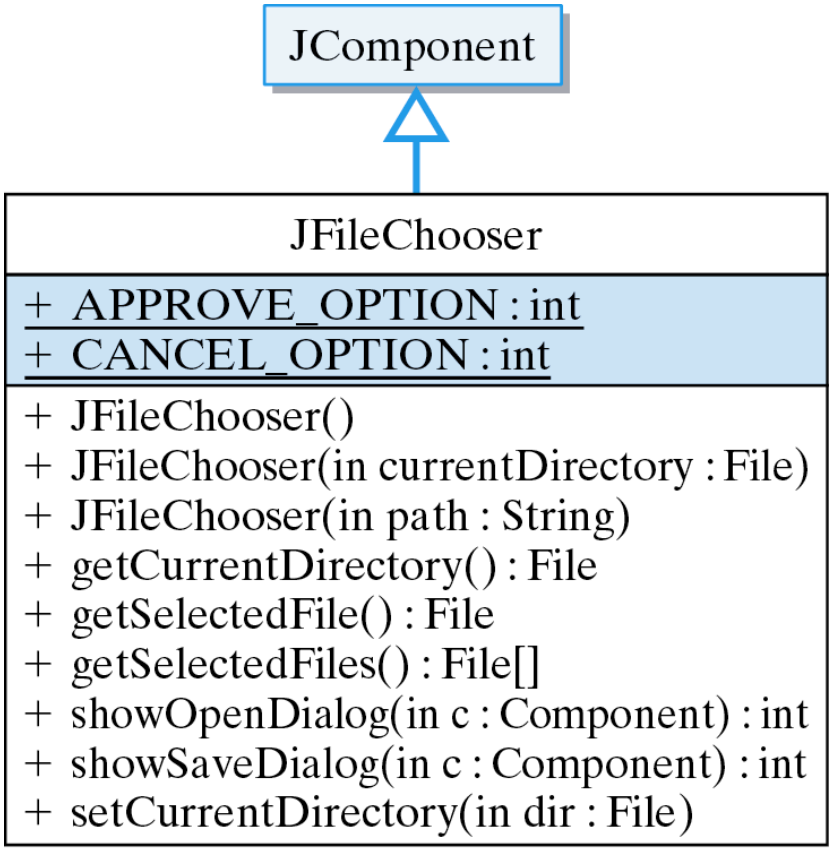
The readRecords() Method

```
private void readRecords(String fileName) {
     try {
         FileInputStream inStream = new FileInputStream(fileName);
         display.setText("Name\tYear\tGPR\n");
         try {
             while (true) {                                  // Infinite loop
                 Student student = new Student();    // Create a student
                 student.readFromFile(inStream);      //  and read its data
                 display.append(student.toString() +  "\n");
             }
         } catch (IOException e) {                          // Until IOException
         }
         inStream.close();
     } catch (FileNotFoundException e) {
         display.append("IOERROR: File NOT Found: " + fileName + "\n");
     } catch (IOException e) {
         display.append("IOERROR: " + e.getMessage() + "\n");
     } catch (ClassNotFoundException e) {
         display.append("ERROR: Class NOT found " + e.getMessage() + "\n");
     }
} // readRecords()
```

Deserialization

Note: readToFile() was defined before.

JFileChooser Class



Example: Opening a File

```
JFileChooser chooser = new JFileChooser();
int result = chooser.showOpenDialog(this);

if (result == JFileChooser.APPROVE_OPTION) {
    File file = chooser.getSelectedFile();
    String fileName = file.getName();
    display.setText("You selected " + fileName);
} else
    display.setText("You cancelled the file dialog");
```

Display the dialog.

Interpret the user's action.

Technical Terms

- absolute path name
- binary file
- buffer
- buffering
- database
- data hierarchy
- directory
- end-of-file character
- field
- file
- filtering
- input
- object serialization
- output
- path
- record
- relative path name
- Unicode Transformation Format (UTF)

Summary Of Important Points

- A *file* is a collection of data stored on a disk.
- A *stream* is an object that delivers data to and from other objects.
- An InputStream (e.g., System.in) is a stream that delivers data to a program from an external source.
- An OutputStream (e.g., System.out) is a stream that delivers data from a program to an external destination.
- The java.io.File class provides methods for interacting with files and directories.

Summary Of Important Points (cont)

- The *data hierarchy*: a *database* is a collection of files. A *file* is a collection of records. A *record* is a collection of fields. A *field* is a collection of bytes. A *byte* is a collection of 8 bits. A *bit* is one binary digit, either 0 or 1.
- A *binary file* is a sequence of 0's and 1's that is interpreted as a sequence of bytes. A *text file* is a sequence of 0s and 1s that is interpreted as a sequence of characters.

Summary Of Important Points (cont)

- *Buffering* is a technique in which a temporary region of memory (*buffer*) is used to store data during input or output.
- A text file is divided into lines by the \n character and ends with a special *end-of-file* character.
- Standard *I/O algorithm*: (1) Open a stream to the file, (2) perform the I/O, (3) close the stream.
- Most I/O methods generate an IOException.

Summary Of Important Points (cont)

- Effective *I/O design*: (1) What streams should I use to perform the I/O? and (2) What methods should I use to do I/O?
- Text input methods return null or -1 when they encounter the *end-of-file* character.
- Binary read methods throw EOFException when they read past the end of the file.
- Object *serialization/deserialization* is the process of writing/reading an object to/from a stream.