

# JAV

## Spring 2014

### **Lecture 4**

*Java I/O*  
*Console and File*

# Lecture outline

- Console I/O
- File I/O

# Console I/O

- Console output
- Console input

# Console output

- Handled by `System.out` member
  - an object of `java.io.PrintStream`
- Output methods:
  - `print`
  - `println`
  - `printf`

# print/println

- `System.out.print`: prints without end-of-line
- `System.out.println`: prints with end-of-line
- Methods of `System.out` for writing data values to the output console
- Overloaded to support primitive and object-type arguments
- For object-type args, `toString()` is used
- Supports in-line string concatenation (+)

# printf

- [Java 5.0] `System.out.printf`
- Print formatted output
- Takes one or more arguments:
  - Format string arg
  - Value args (optional): values to be outputted
- Format string: consists of
  - texts (optional) and
  - one or more *format specifiers*, one per argument

# Program PrintfBasicDemo (1)

```
String s = "price is: ";
```

```
System.out.printf("%n");
```

```
System.out.printf(s);
```

format  
string

```
System.out.printf("%s", s);
```

format  
string &  
values

```
// on the same line as above
```

```
double price = 19.8d;
```

```
System.out.printf("%6.2f", price);
```

# PrintfBasicDemo (2)

```
// on a new line
```

```
System.out.printf("%n%6.2f%n", price);
```

```
// print both output values
```

```
System.out.printf("%s%n%6.2f", s, price);
```



# Format specifier (1)

- Specifies the format of one output argument
- Basic syntax: `%[arg_index][l][m.n]c`

`%`: the format marker

*arg\_index*: the argument index

*m*: (optional) the field width or number of spaces used for output

*c*: the conversion character

# Format specifier (2)

n: (optional) the number of digits after the decimal point

l: (optional) flag (e.g. output alignment)

(empty)	right justified
---------	-----------------

-	left justified
---	----------------

# Conversion characters

d: decimal integer

f : fixed-point floating point

e: E-notation floating point

g: general floating point

s: string

c: character

b: boolean

?: percentage

n: line break

# Program PrintfDemo (1)

What do these do?

```
String aString = "abc";  
System.out.printf("%4s %n", aString);  
  
char ch = 'Z';  
System.out.printf("%4c %n", ch);  
System.out.printf("%-4c %n", ch);
```

# PrintfDemo (2)

What do these do?

```
double d = 12345.123456789;  
System.out.printf("%.4f %n", d);  
System.out.printf("%12.4f %n", d);  
System.out.printf("%-12.4f %n", d);  
System.out.printf("%12.5e %n", d);  
System.out.printf("%-12.5e %n", d);
```

# PrintfDemo (3)

What do these do?

```
double d = 20.123
```

```
System.out.printf("%.0f%% %n", f);
```

# PrintfDemo

`io.ch2.PrintfDemo`

# Console input

- Read user data from standard input
- Use class Scanner



# java.util.Scanner

- [Java 1.5]
- Scans text and primitive types from a source
- Breaks input into tokens based on a configurable delimiter (default is space)
- Tokens are primitive (e.g. `int`, `long`, etc.) or `String`-type
- Tokens can be retrieved using `nextX()` methods

# Scanner methods

- `nextInt`: reads and returns the next token as integer type
- `nextDouble`: reads and returns the next token as a double floating point number
- `next`: reads and returns the next token as a word
- `nextLine`: reads and returns the rest of the current line (*excluding EOL character*)
- `useDelimiter`: sets the delimiter pattern

# SimpleScannerDemo

```
String input = "1.0 fish 2 fish red  
fish blue fish";  
String delim = "\\s*fish\\s*";  
Scanner s = new  
    Scanner(input).useDelimiter(delim);  
out.println(s.nextDouble());  
out.println(s.nextInt());  
out.println(s.next());  
out.println(s.next());  
s.close();
```

# String scan example

```
String i2 = "hello world\nto be  
or \nnot to be";  
s = new Scanner(i2);  
out.println(s.next());  
out.println(s.nextLine());  
out.println(s.next());  
out.println(s.nextLine());  
s.close();
```

# Scan user input

- Creates a Scanner object whose source is standard input:

```
Scanner s = new Scanner(System.in)
```

# Program SelfService (1)

```
Scanner keyboard = new Scanner(System.in);  
  
out.println("Enter number of items purchased");  
out.println("followed by the cost of one  
item");  
...
```

# SelfService (2)

```
int count = keyboard.nextInt();  
double price = keyboard.nextDouble();  
double total = count * price;  
  
out.printf("%d items at $%.2f each.  
%n", count, price);  
out.printf("Total amount due $%.2f.  
%n", total);  
out.printf("Place $%.2f in an envelope  
%n", total);
```

# SelfService

`io.ch2.SelfService`



# File I/O

- Overview: stream, file, text and binary files
- Common file I/O programming tasks
- The File class
- Text file I/O
- Object file I/O
- [0!] Random access file I/O

# Stream

- A flow of data between a program and some I/O device or file
- Input stream:
  - an input flow into the program (e.g. from a file or a keyboard)
- Output stream:
  - an output flow from the program (e.g. to a file or the console)
- Streams are objects of some class in package `java.io`

# Stream examples

`java.io.`

`OutputStream:`

- the super class of all output streams

`PrintStream:`

- an output stream for writing data values (e.g. `System.out` and `System.err`)

`InputStream:`

- a super class of all input streams (e.g. `System.in`)

# File

- A common type of data storage for programs
- Often used to store:
  - configuration details
  - a (small) set of data records
  - text data: a sequence of readable characters
  - binary data: chunks of bytes

# Text file

- Contains lines of text marked by end-of-line characters
- Readable by humans (e.g. using a text editor)
- The EOL character differs between host systems

# Binary file

- Contains chunks of bytes, usually not-readable by humans
- But efficient for programs to access (esp. with random access)
- No EOL markers

# File path name

- An absolute path to a file, containing a directory path and a file name
- Used to locate a file on disk
- Current directory is assumed if path is omitted
- Delimited by a path separator:
  - `File.separator`
  - `System.getProperty("file.separator")`

# Examples

`myfile.txt`

`mydir/mysubdir/myfile.txt // Unix`

`mydir\mysubdir\myfile.txt //Windows`

- must be escaped with `"\\` in Java strings



# class java.io.File

- Represents a handle for file or a directory
- Provides methods to:
  - query properties of file
  - operate on file
- Create a File object of a file fileName:  
`File fileObject = new File(fileName);`

# Program FileDemo (1)

```
Scanner keyboard =  
    new Scanner(System.in);  
String fileName = null;  
  
System.out.println("Enter a file name:");  
fileName = keyboard.nextLine();  
  
File fileObject = new File(fileName);
```

# File methods

- Constructor
- Accessors
- Mutators (with side effects)

# Constructor

```
public File(String File_Name)
```

- *File\_Name*: the (absolute or relative) abstract path name of the file

# Accessors

- `exists`: true if the file exists
- `canRead`: true if the file is readable (for file IO)
- `canWrite`: true if the file is writable (for file IO)
- `getName`: returns the name part of the abstract path name (e.g. `myfile.txt`)
- `getPath`: returns the path part of the abstract path name (e.g. `io`)
- `isFile`: true if the object is a file
- `isDirectory`: true if the object is a directory
- `length`: returns the file size in bytes

# Mutators (with side effects)

- `setReadOnly`: sets the file to read-only
- `setReadable`: sets the file to readable (for file IO)
- `setWritable`: sets the file to writable (for file IO)
- `setExecutable`: sets the file to executable
- `delete`: delete the file on disk
- `createNewFile`: create a new file on disk
- `mkdir`: make a new directory on disk

## Program FileDemo (2)

```
while (fileObject.exists()) {  
    System.out.println("There already  
        is a file named " + fileName);  
    System.out.println("Enter a  
        different file name:");  
    fileName = keyboard.nextLine();  
    fileObject = new File(fileName);  
}
```

# FileDemo (3)

```
// create the file
try {
    fileObject.createNewFile();

    // display file properties
    System.out.println("File created:
" + fileName);

    System.out.println("name: " +
fileObject.getName());
```



# FileDemo (4)

```
System.out.println("absolute path: " +  
fileObject.getAbsolutePath());
```

```
System.out.println("path: " +  
fileObject.getPath());
```

```
...
```

```
System.out.println("size: " +  
fileObject.length());
```

# FileDemo (5)

```
// delete the file
if (fileObject.delete()) {
    System.out.println("File deleted: " +
        fileName);
} else {
    System.err.println("Failed to delete
        file: " + fileName);
}
} catch (IOException e) { ...}
```

# Program FileDemo

`io.ch10.FileDemo`

# File I/O programming tasks

- Create a file handle using `File` class
- Create a stream object to read/write from/to file
- Perform file operations using the stream object
- Close stream object when finished

# Which stream object?

- Depends on the file operation and data type
- Operation:
  - read: input stream(s)
  - write, append: output stream(s)
- Data type:
  - text: character streams
  - binary: byte streams
  - object: object streams

# Text file I/O

- Use character streams
- Supported operations:
  - Write text to a file
  - [!] Append text to a file
  - Read text from a file

# Write text to file

- `FileOutputStream`
  - The byte output stream for writing to a file
  - Wraps around the `File` object (handle)
  - Optional for simple write operation
  - Required for append operation
- `PrintWriter`
  - Represents the character stream for writing text to file
  - Wraps around a `FileOutputStream` or `File` object

# TextFileOutputDemo2 (1)

```
File f = null;  
try {  
    // create File object  
    f = new File("stuff.txt");  
    // create output stream object  
    outputStream = new PrintWriter(f);  
} catch (FileNotFoundException e) {  
    . . .  
}
```



# TextFileOutputDemo2 (2)

```
// write to stream  
System.out.println("Writing to  
file.");  
    outputStream.println("The quick  
brown fox");  
    outputStream.println("jumped  
over the lazy dog.");  
  
// close stream  
outputStream.close();
```

# TextFileOutputDemo2

`io.ch10.TextFileOutputDemo2`

# [!] Append text to a file

- Similar to file writing except:
  - using a `FileOutputStream` object to wrap around the file handle
  - specifying `true` as argument

```
outputStream = new PrintWriter(  
    new FileOutputStream(f, true));
```

# [!] TextFileAppendDemo

`io.ch10.TextFileAppendDemo`

# Read text from file

- `FileInputStream`
  - Wrap around the file object, but not required
- `java.util.Scanner`
  - Reads word(s) at a time or a line at a time
  - Uses a configurable delimiter to parse input
- `[!] BufferedReader`
  - Reads a line at a time

# TextFileScannerDemo2 (1)

```
Scanner inputStream = null;
File f = null;
try {
    // create the file object
    f = new File("morestuff.txt");
    // create the stream object
    inputStream = new Scanner(f);
} catch (FileNotFoundException e) {
    . . .
}
```

# TextFileScannerDemo2 (2)

```
// read file
int n1 = inputStream.nextInt();
int n2 = inputStream.nextInt();
int n3 = inputStream.nextInt();

inputStream.nextLine();
String line = inputStream.nextLine();
// close stream
inputStream.close();
```

# TextFileScannerDemo2

`io.ch10.TextFileScannerDemo2`



# Object file I/O

- To read/write objects from/to file
  - in binary format
- Use object streams
- The object type (class) must implement `java.io.Serializable` interface

# Serializable class

- Object type must implement `java.io.Serializable`
  - default: no method implementation is required
  - enhanced: involves some implementation
- Example:

```
import java.io.Serializable;

public class SomeClass implements
    Serializable {

    . . .

}
```

# Writing objects

- To create the output stream:
  - create a `FileOutputStream` object
  - create an `ObjectOutputStream` from file stream
  - write objects using method `writeObject(Object)`
- Example:

```
SomeClass o = new SomeClass(1, 'A');  
outputStream.writeObject(o);
```

# Reading objects

- To create the input stream:
  - create a `FileInputStream` object
  - create an `ObjectInputStream` from file stream
  - Read objects using method `readObject()`
  - Cast object to the declared type

- Example:

```
SomeClass obj = (SomeClass)  
                inputStream.readObject();
```

# ObjectIODemo2 (1)

```
// create output file
File f = new File("datafile");

try {
    // create output streams
    ObjectOutputStream outputStream =
        new ObjectOutputStream(
            new FileOutputStream(f));
```

# ObjectIODemo2 (2)

```
SomeClass oneObject =  
    new SomeClass(1, 'A');  
// write object  
outputStream.writeObject(oneObject);  
// close stream  
outputStream.close();  
} catch (IOException e) {  
    System.out.println("...");  
}
```

# ObjectIODemo2 (3)

```
try {  
    // create input stream objects  
    ObjectInputStream inputStream =  
        new ObjectInputStream(  
            new FileInputStream(f));  
  
    // read objects  
    SomeClass readOne = (SomeClass)  
        inputStream.readObject();  
}
```

# ObjectIODemo2 (4)

```
// close stream
InputStream.close();
} catch (FileNotFoundException e) {
    System.out.println("...");
} catch (ClassNotFoundException e) {
    System.out.println("...");
} catch (IOException e) {
    System.out.println("...");
}
```



# ObjectIODemo2

`io.ch10.ObjectIODemo2`

# ObjectIODemo3

`io.ch10.ObjectIODemo3`

- Use a loop to read objects
  - explained later in `ObjectCustomIODemo`
- [!] Can also use a loop to write objects

# Custom object I/O

- Customise how objects of a serializable class are stored:
  - serial version UID
  - read/write operations

# Serial version UID

- A unique class version number
- Used during deserialisation to verify the object type
- Automatically computed by JVM for each serializable class
- Should be explicitly declared:

```
private static final long  
serialVersionUID = Long_Number;
```

# Serialisation operations

- Implementations of read/write operations can be changed
- For object writing:

```
private void writeObject(ObjectOutputStream  
out) throws IOException
```

- For object reading:

```
private void readObject(ObjectInputStream in)  
throws IOException, ClassNotFoundException
```

# Serialising static variables

- Static variables are not serialized by default
- Customise read/write to serialise them
- To write:
  - use `defaultWriteObject()` & `writeX()`
  - (X matches the data type of variable)
- To read:
  - use `defaultReadObject()` & `readX()`
  - (X matches the data type of variable)

# SomeClass2 (1)

```
public class SomeClass2 implements  
Serializable {
```

```
    private static final long
```

```
        serialVersionUID = 2012L;
```

```
    private int number;
```

```
    private char letter;
```

```
    // a static member
```

```
    private static long counter;
```

## SomeClass2 (2)

```
private void writeObject(ObjectOutputStream out)
                        throws IOException {
    // invoke default write first
    out.defaultWriteObject();

    // now write static variables
    out.writeLong(counter);
}
```



## SomeClass2 (3)

```
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException {
    // invoke default read first
    in.defaultReadObject();

    // read static variables
    this.counter = in.readLong();
}
```

# ObjectCustomIODemo (1)

```
try { // WRITE
    ObjectOutputStream outputStream = new
        ObjectOutputStream(
            new FileOutputStream(f));
    char c = 'M';
    for (int i = 0; i < 5; i++) {
        SomeClass2 oneObject =
            new SomeClass2(1+i, (char) (c+i));
        outputStream.writeObject(oneObject);
    }
    outputStream.close();
}
```

# ObjectCustomIODemo (2)

```
try { // READ
    ObjectInputStream inputStream =
        new ObjectInputStream(
            new FileInputStream(f));
    SomeClass2 o;
    try {
        while (true) {
            o = (SomeClass2) inputStream.readObject();
            System.out.println(o);
        }
    } catch (EOFException e) {
        // end of file, finished reading
    }
    inputStream.close();
}
```

# ObjectCustomIODemo

`io.ch10.ObjectCustomIODemo`

# Summary

- Java performs all I/O operations via file handles and streams
- A file handle is represented by `java.io.File`
- `Console` uses `PrintStream` to write normal or formatted output
- `Console` uses `Scanner` to read user input from keyboard
- File I/O use streams specific to the content type and operation (write/append)
- Object I/O uses object streams to r/w objects

# References

Savitch W., Absolute Java, 4th, Addison-Wesley, 2009

- Chapter 2, 10