

# Anonymous Classes for File & Sorting (Ch 4.3 – 4.5)

# Java Odds & Ends (For Assignment 2)

# Command Line

- Compile your project and open a terminal / command line to:  
..
- **Run:** `java <package>.<class> your args here`

```
package ca.cmpt213.as1content;
public class DemoArguments {
    public static void main(String[] args) {
        // Print all arguments
        for (String arg : args) {
            System.out.println("Argument: " + arg);
        }
    }
}
```

```
C:\...\daProject\out\production\daProject> java ca.cmpt213.as1content.DemoArguments Hello world!
Argument: Hello
Argument: World!
C:\...\daProject\out\production\daProject>
```

# Formatted Printing

- Use `printf()` to print formatted numbers:  
`System.out.printf(<format string>, <arg0>, ...);`
  - Format String:..
  - Arguments: Extra data to print.
- Example:  
`System.out.printf("%s! Is it %b that you're %d?%n",  
"Waldo", true, 42);`

## Common Conversion Specifiers

<code>%d</code>	decimal (int)
<code>%x</code>	hexadecimal
<code>%f</code>	float
<code>%s</code>	String
<code>%b</code>	boolean
<code>%n</code>	new line (like <code>\n</code> )

# Formatted Printing (cont)

- **Formatting floats and columns**

- Round to 2 decimal-point places:...
- Use at least 5 columns to print:...
- Print with comma groupings:...

- **Examples**

```
double a = 154.7599;
```

```
int b = 98765431;
```

```
System.out.printf("Values: %,15.2f,  %,5d%n", a, b);
```

```
Values:      154.76,   98,765,431
```

- **PrintWriter Note**

- Using **PrintWriter** to write to **System.out**, call its **flush()** method when done output.

# Wrappers & Shuffle

- Primitive data types cannot be use when you need a class (such as in an **ArrayList**).
  - **Wrapper**...
  - Java has immutable wrappers for primitive data types: **Integer**, **Double**, **Boolean**, **Character**, etc

- **Example:**

```
// Create the ArrayList
```

```
List<Double> values = new ArrayList<>();
```

```
// Make a Double wrapper object from the double value.
```

```
values.add(new Double(6));
```

```
values.add(new Double(0));
```

```
values.add(4);
```

```
// Shuffle (generate a random permutation):
```

```
java.util.Collections.shuffle(values);
```

Can be done without  
**new Double(4)**

..

# File, FileFilter and Anonymous Classes

# File Class

- File Access

- Use the File class to work with file names:

`File file = new File("C:/t/file.txt");`

- Useful methods:

- Get the path
  - Does the file exist?
  - Get it's size in bytes..
  - Is it a directory?
  - Get all files in the folder..

`file.getAbsolutePath()`

`file.exists()`

`file.isDirectory()`



# FileFilter

- Making `listFiles()` filter
  - We need to tell `listFiles()` what type of files we want.
  - Let's write a method it can call to ask us (for each file) if we want to accept it:..
- Interface
  - An interface is..
- Java puts `accept()` into an interface

```
public interface FileFilter {  
    boolean accept(File pathName);  
}
```

# Using FileFilter

- Process to use FileFilter:

- 1) Write a custom-filter class which..

(Similar to inheritance).

- 2) Instantiate our custom-filter.

- 3) Pass our custom-filter to File's listFiles() function.

- 4) Use the results!

# Anonymous Classes

- Anonymous class:
- Useful when you need a short custom class to..
  - custom sorting
  - filtering files in a list
  - a button's callback

- Generic Example

```
public static void main(String[] args) {  
    ClickHandler buttonAction = new ClickHandler() {  
        @Override  
        public void handleClick(){  
            System.out.println("Clicked!");  
        }  
    };  
    setButtonCallback(buttonAction);  
}
```

**ClickHandler**  
is the interface  
(fictitious).

Use IDE to add  
  
to the anonymous  
class.  
**(IntelliJ: Alt-Enter)**

# Use a anonymous FileFilter

```
private static void demoFileFilter() {  
    // Create the filter (an anonymous class)  
    FileFilter filter = new FileFilter() {  
        @Override  
        public boolean accept(File file) {  
            return file.getName().endsWith(".txt");  
        }  
    };  
  
    // Use the filter (with callback)  
    File folder = new File("C:/t/");  
    File[] fileList = folder.listFiles(filter);  
  
    for (File subFile : fileList) {  
        System.out.println(" sub file: " + subFile.getAbsolutePath());  
    }  
}
```

Note the ()

# Anonymous Object & Class

- Anonymous Object:..
- Anonymous Class:..

```
private static void demoFileFilter() {  
    File folder = new File("C:\\t\\");  
  
    // Create filter (anonymous object of an anonymous class)  
    File[] fileList = folder.listFiles(new FileFilter() {  
        @Override  
        public boolean accept(File file) {  
            return file.getName().endsWith(".txt");  
        }  
    });  
  
    for (File subFile : fileList) {  
        System.out.println(" sub file: " + subFile.getAbsolutePath());  
    }  
}
```

Note the });

# Sorting with Comparable

# Sorting

- Java & Sorting

- built-in sorting for collection: arrays, **ArrayList**, etc.
- Calling Java's sort method for collections:  
**java.util.Collections.sort( myCars );**
- Elements in the collection must implement the **Comparable** (generic) interface:

```
interface Comparable<Type> {  
    // Compare this object with the specified object returning  
    //      negative integer for      this < obj  
    //      zero for                  this == obj  
    //      positive integer for      this > obj  
    int compareTo(Type obj);  
}
```

# Sorting Example

```
public static void main(String[] args) {  
    // Create the list with some items:  
    ArrayList<Pen> list = new ArrayList<Pen>();  
    list.add(new Pen("Green", 14));  
    list.add(new Pen("Orange", 20));  
    list.add(new Pen("Blue", 75));  
  
    // Sort the list  
    java.util.Collections.sort(list);  
  
    // Output the list.  
    for (Pen item : list) {  
        System.out.println(item);  
    }  
}
```

## Output:

```
Pen [Blue, 75%]  
Pen [Green, 14%]  
Pen [Orange, 20%]
```

```
class Pen implements Comparable<Pen> {  
    String colour;  
    int filled;  
    // ... Some content omitted...  
  
    @Override  
    public int compareTo(Pen other) {  
        return colour.compareTo(  
            other.colour);  
    }  
}
```



# Notes on sort

- **Comparable** interface defines the..
  - This is the one order which you choose as the default order for your class.
- **java.util.Collections.sort()** method does:
  - Copies all elements into an array,
  - Sorts the array,
  - Copies each element back into the original data type
- **Guaranteed “fast” sort**
  - $O(n \log(n))$  performance (which is good)

# Sorting with Comparator

# Multiple Sort Orders

- What about sorting by a number of different orders?
  - The **Comparable** interface only allows us to define..
  - What if I want to sort **Pens** by **colour**, or by **filled %**?
- Must create a **Comparator**:
  - Create an extra little class which implements a custom comparison function.
  - This class implement the **Comparator** interface.
  - We create an instance of this class when sorting.

# Comparator Interface

- Comparator interface:
  - Used by sort algorithms.
  - It's a generic type: so you specify a type.

```
interface Comparator<Type> {  
    // Compare 2 objects for custom order.  
    // Returns:  
    //     negative integer for      o1 < o2  
    //     zero for                  o1 == o2  
    //     positive integer for      o1 > o2  
    int compare(Type o1, Type o2);  
}
```

# Implement Comparator

- Make a new class which has one purpose:
  - Implement `compare()` to give the special sort order.

```
class PenSortByFilled implements Comparator<Pen> {  
    // Return a negative number if o1 < o2  
    // Return 0 if equal.  
    // Return a positive number if o1 > o2.  
    @Override  
    public int compare(Pen o1, Pen o2) {  
        return o1.getFilled() - o2.getFilled();  
    }  
}
```

- Call `sort()` by passing an instance of this class:  
`java.util.Collections.sort(list, new PenSortByFilled());`

# Sorting Example with Comparator

```
public static void main(String[] args) {  
    // Create the list with some items:  
    ArrayList<Pen> list = new ArrayList<Pen>();  
    list.add(new Pen("Green", 14));  
    list.add(new Pen("Orange", 20));  
    list.add(new Pen("Blue", 75));  
  
    // Sort the list  
    Collections.sort(list, new PenSortByFilled());  
  
    // Output the list.  
    for (Pen item : list) {  
        System.out.println(item);  
    }  
}
```

## Output:

```
Pen [Green, 14%]  
Pen [Orange, 20%]  
Pen [Blue, 75%]
```

# Strategy Pattern

- **FileFilter & Comparator**
  - Each defines a special purpose class to..
  - Often used as anonymous classes, and anonymous objects.
  - These are examples of the..
- **Strategy Pattern**
  - 
  - The algorithm (in our anonymous classes) can change without changing the general function (`java.util.Collections.sort()`).

# Summary

- Formatted printing with **printf()**: %n, %d, %f, ...
- **Wrappers**: Turn primitives into objects.
  - **Double**, **Integer**, **Boolean**, **Character**
- **File**: For working with files
  - **FileFilter** interface for filtering files.
- **Sorting**
  - Natural order (single order): **Comparable**
  - Custom order (many orders): **Comparator**
- **Anonymous classes & objects**
  - Example of the **Strategy Pattern**.