Topic 2: Anonymous Inner Classes

Interface

Interfaces (1)

- An *interface* is a class that has all abstract methods.
  - It cannot be instantiated, and
  - all of the methods listed in an interface must be written elsewhere.
- The purpose of an interface is to specify behavior for other classes.
- It is often said that an interface is like a “contract,” and when a class implements an interface it must adhere to the contract.
- An interface looks similar to a class, except:
  - the keyword interface is used instead of the keyword class, and
  - the methods that are specified in an interface have no bodies, only headers that are terminated by semicolons.

Interfaces (2)

- The general format of an interface definition:

```
public interface InterfaceName
{
  (Method headers...)
}
```

- All methods specified by an interface are public by default.
- A class can implement one or more interfaces.

Interfaces (3)

- If a class implements an interface, it uses the `implements` keyword in the class header.

```
public class FinalExam implements
    GradedActivity
```

- Example:
    - (Person.java)
    - (InterfaceDemo.java)
    - (Displayable.java)

File and FileFilter

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:

FileFilter

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

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

Using FileFilter

- The process to use FileFilter:
  1. Write a custom-filter class which implements the FileFilter interface.
  2. Instantiate our custom-filter.
  3. Pass our custom-filter to File's listFiles() function.
  4. Use the results!

(TxtFilter.java)

(DemoFileFilter.java)

Nested Classes

And Lambda expressions

Nested Classes

- A nested class is one that is defined within another class.
- In Java, nested classes can be static or non-static
  - Concept is very similar to static and non-static methods and properties.
  - A static nested class does not need a reference to the containing class. It can be used in a static context
  - A non-static class (or simple inner class) requires a reference to the to its containing class. It can be classified as one of the following three groups:
    1. Member class
    2. Local Inner class
    3. Anonymous Inner class

(Outer.java)

Anonymous Inner Classes

- An inner class is a class that is defined inside another class.
- An anonymous inner class is an inner class that has no name.
- An anonymous inner class must implement an interface, or extend another class.
- Useful when you need a class that is simple, and to be instantiated only once in your code.

Anonymous Inner Classes

- Let’s revisit the file filter.
- Since the TxtFilter class is only used by the DemoFileFilter class (and no other class), we can bypass naming the TxtFilter
- We can do this in one of two ways:
  - Use an anonymous class
  - Use an anonymous object

(DemoFileFilter.java)

Functional Interfaces and Lambda Expressions

- A functional interface is an interface that has one abstract method.
- A lambda expression can be used to create an object that implements the interface, and overrides its abstract method.
- In Java 8, these features work together to simplify code, particularly in situations where you might use anonymous inner classes.

Comparable and Comparator

… and Sorting

Sorting

- Java has built-in sorting for collections: 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);
}
```

(Person.java)

(SortPerson.java)

Java Sorting

- Comparable interface defines the natural order
- This is the one order which you choose as the default order for your class.
- `java.util.Collections.sort()` method:
  - Copies all elements into an array,
  - Sorts the array using a guaranteed "fast" sort,
  - Copies each element back into the original data type
  - O(n log(n)) performance

Multiple Sort Orders

- What about sorting by a number of different orders?
- The Comparable interface only allows us to define one sort order.

- If we would like more than one sort order, 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

- The comparator interface is used for special comparator objects.

```
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);
}
```

- To implement comparator, we make a new class which has one purpose:
- Implement `compare()` to give the special sort order.
- Call `sort()` by passing an instance of this class:
  `java.util.Collections.sort(list, comparatorObject);`

(SortPerson.java)