DEPAUL UNIVERSITY
COLLEGE OF COMPUTING AND DIGITAL MEDIA

CSC 281 NOTES
Spring 2019: Amber Settle

Wednesday, May 29, 2019

Announcements

- Discuss the seventh assignment
- The eighth assignment is due tonight – questions?
- The ninth assignment is due Wednesday, June 5th – it will be the last graded assignment

Arrays class

Java provides a class called **Arrays** that allows you to manipulate arrays in various ways, for example by sorting the array or by searching the array for a value.

Some of the methods of interest include (where T is one of double, int, char, float, long, Object, short, etc.):

- **static int binarySearch(T [] a, T key)**: Find the index of key in the array a using binary search
- **static boolean equals(T[] a, T[] a2)**: Return true if a and a2 contain the same elements in the same order and false otherwise
- **static void sort(T[] a)**: Sort the array a into ascending order, where ascending order is defined by the type T – uses quicksort
- **static String toString(T[] a)**: Return a String representation of the array a

You can see some example uses of the Arrays class in the file **UseArrays.java**.

Review: Comparable interface

Classes implement the Comparable interface to allow objects to be ordered. The classes Integer, Double, and String, as well as many others, implement the interface.

The Comparable interface has one method:

public int compareTo(Object object)

which returns a negative integer if the calling object is less than the argument, zero if the two objects have equal values, and a positive integer if the calling object is larger than the argument.

Note: String, Double, and Integer are all classes that implement the compareTo() method. As such, they are considered to implement the Comparable interface.

The Comparable interface has a type associated with it. This means that the header for the Comparable interface looks like:

public interface Comparable<T>

When we want to alter our method to use the Comparable interface, we must include the type T in our header (in a somewhat elaborate way which is shown in the examples below). The type T is what we then refer to inside our methods when we want to discuss the type of our objects.

Note: The **Arrays.sort()** method is defined to use the compareTo() method unless you specify otherwise.

The Comparator interface

Suppose you want to sort an array of Strings using the length of the String rather than the lexicographic order of the String.

The String class can only implement compareTo() a single way, and we don't have the ability to modify the String class.

To deal with this situation **there is a second version of the Arrays.sort() method whose parameters are an array and a comparator**, that is, an instance of a class that implements the Comparator interface:

```
public interface Comparator<T> {
      int compare(T first, T second)
}
```

To implement this interface you have to import it from java.util.

Then **to compare strings by length** you define a class that implements the `Comparator<String>`, create an instance of it, and then pass one of those objects to the Arrays.sort() method.

See the file **StrLenComp.java** for the class that implements a String comparator, and the file **StrCompare.java** for the code that uses the new class.

Using lambdas for Comparators

A **lambda expression** is a block of code that you pass around so it can be executed later, once or multiple times.

For example, if we want to sort using the Comparator interface, we need to create a method to pass to Arrays.sort() to get that done.

Let's look at the situation of **comparing strings by length** to form our first lambda expression.

We want to write `first.length – second.length`, but we need a way in Java to say that first should be a String object and second should be a String object.

So we can write:

```
(String first, String second) -> first.length() –
second.length()
```

In general **a lambda expression is a block of code, together with the specification of any variables that must be passed to the code**.

If the body of a lambda expression doesn't fit into a single expression, you write it enclosed in curly braces and with explicit return statements.

For example:

```
(String first, String second) -> {
      int diff = first.length() – second.length();
      if (diff < 0) return -1;
      else if (diff > 0) return 1;
      else return 0;
}
```

If the **lambda has no parameters** then you provide empty parentheses.

We can re-write the string comparator example using a lambda which eliminates the need for the Comparator class. Put the modified example into the **AltStrCompare.java** file.

Nested classes

You can place a class inside of another class.

Such a class is called a **nested class**.

Doing this allows you to avoid cluttering up a package with objects that will be used only in a single class.

There are **two ways to write nested classes**:

1. As a static class
2. As an inner (non-static) class

We'll consider examples of both.

Static nested classes

Typically when you create a nested class it is for the use of the enclosing class only.

In that case you can make the class private. You then don't need to worry about making the instance variables private since the class can't be accessed from outside the enclosing class.

Example: An Invoice class that bills for items, each of which has a description, quantity, and unit price. See the first version in **Invoice.java** and **UseInvoice.java**.

You can make a nested class public if you like. In that case you want to use standard encapsulation techniques such as making instance variables private.

In that case you can use the nested class as you would any other class, as long as you write the name of the enclosing class in front of it.

Example: Make the following modifications to the invoice example:

- Make the Item class public and all of its instance variables private.
- Add constructors so that the values can be set from outside the class.
- Create an Item object in the main method.

Inner classes

With inner classes you don't make the nested class static, which has implications for the way that the class can be used.

Let's illustrate this using an example.

Consider a social network in which the network class has an inner class representing members of the social network.

Each member of the social network has a list of friends who are connected with him/her in the network.

See the **Network.java** file for the classes and **UseNetwork.java** for the use of the classes.

With the static modifier dropped, **the inner class can access instance variables of the outer class** just by using their name.

> **Example**: See the **remove**() method in the Member class. It can reference the list of members in the Network class by using the name of the instance variable.
>
> Note that it can also access methods of the outer class in the same way. There aren't any examples in the provided code, but it works the same way as the instance variables.

A nested static class does not have this ability. So we used inner classes when having the ability to modify the enclosing class is useful or desired.

Although you don't need to reference the enclosing object in the example we saw above, it can be useful at times to have the ability to do so.

To access the outer object that is enclosing the inner object, you write:

```
nameOfOuterClass.this
```

where `nameOfOuterClass` is the name of the outer class.

So in our example, we would write:

```
Network.this
```

You can see an example of this in the **createdIn() method of the Network class**. View the **Network.java** file for the code.