Object-Oriented Programming

Interfaces

Interfaces

- In object-oriented programs using Java, we use interfaces to specify a set of behaviors that a number of object classes should have in common.
- In Java, if class B **`implements`** interface A, then B must provide implementations of all method signatures given in A.
- Interface A does not contain any instance variables.
- Interface A only contains signatures of methods that must be implemented.

Example: `Comparable`

```
public interface Comparable
{
     int compareTo(Object obj);
}
```

This interface specifies one method that must be implemented by each class that **`implements Comparable`**. (All signatures are **`public`**.)

More about interfaces

- Java uses interfaces to provide a consistent way of presenting common behavior amongst classes that are of different types.
- Every class that implements the interface must allow its users to call the implemented methods in the same way, regardless of the class.

The `Comparable` interface

`int compareTo(Object obj)`

- Compares this object with the specified object [given as the parameter] for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

- The **`String`** class implements **`Comparable`**
- Therefore, it must have a **`compareTo`** method with the signature given above.
  - The implementation should also follow the description above, but the compiler can't check this explicitly.

The String class revisited

Look at the Java API for **`String`**.

- implements Comparable
- has the following method:

  `public int compareTo(Object obj)`

  Compares this string with the given object (assuming it is a string) lexicographically.

  Returns 0 if this string is lexicographically equal to the given string.

  Returns a value less than 0 if this string is lexicographically less than the given string.

  Otherwise returns a value greater than 0.

Lexicographical ordering

Similar to alphabetical ordering, except we include digits and other punctuation.

General lexicographic rule of thumb:

- digits come before uppercase letters
- uppercase letters come before lowercase letters
- Example: Lexicographic ordering
  - 01234
  - 012DE
  - ABCDE
  - ABcDe
  - abcde

If strings only have letters (upper- or lower-case, not both) and possibly spaces, lexicographical ordering reduces to alphabetical ordering.

Using `compareTo` with strings

```
public String getFirstCity(String[] cityArray)
{
  // find first city alphabetically in array
  int firstCity = cityArray[0];
  for (int i=1; i<cityArray.length; i++) {
      if (cityArray[i].compareTo(firstCity) <
  0)
          firstCity = cityArray[i];
  }
  return firstCity;
}
```

Example: `Date`

- A (calendar) date consists of
  - **`month`** - an integer between 1 and 12, inclusive
  - **`day`** - an integer between 1 and 31, inclusive
  - **`year`** - an integer
- Suppose **`Date`** is defined as follows:

```
public class Date implements Comparable {
...
}
```

  - The compiler will force us to write a **`compareTo`** method to satisfy the interface definition.

`compareTo` for `Date`

```
public class Date implements Comparable
{
  public int compareTo(Object obj)
  {
   Date other = (Date) obj;
   if (this.year != other.year)
     return this.year - other.year;
   else if (this.month != other.month)
     return this.month - other.month;
   else
     return this.day - other.day;
  }
```

`equals` for `Date`

```
  public boolean equals(Object obj)
  {
   return (this.compareTo(obj) == 0);
  }

  // other methods not shown

} // end Date class
```

Selection Sort Algorithm

- Traverse the array for the minimum value.
- Swap this value with the value in cell 0.
- Traverse the array again (starting from cell 1) for the minimum value.
- Swap this value with the value in cell 1.
- Traverse the array again (starting from cell 2) for the minimum value.
- Swap this value with the value in cell 2.
- Continue this process until the array is completely sorted.

Selection Sort Algorithm

23	97	81	62	18	Find min
18	97	81	62	23	Swap
18	97	81	62	23	Find min
18	23	81	62	97	Swap
18	23	81	62	97	Find min
18	23	62	81	97	Swap
18	23	62	81	97	Find min
18	23	62	81	97	Swap
					Done (why?)

Selection Sort Algorithm
on an array of `int`

```
public static void selectionSort(int[] list) {
  int minPos;
  int temp;
  for (int index = 0; index < list.length-1; index++)
  {
   minPos = index;
   for (int pos = index+1; pos < list.length; pos++)
       if (list[pos] < list[minPos])
            minPos = pos;
   temp = list[minPos];
   list[minPos] = list[index];
   list[index] = temp;
  }
}
```

Selection Sort Algorithm
on an array of `String`

```
public static void selectionSort(String[] list) {
  int minPos;
  String temp;
  for (int index = 0; index < list.length-1; index++)
  {
   minPos = index;
   for (int pos = index+1; pos < list.length; pos++)
       if (list[pos].compareTo(list[minPos]) < 0)
            minPos = pos;
   temp = list[minPos];
   list[minPos] = list[index];
   list[index] = temp;
  }
}
```

Selection Sort Algorithm
on an array of `Date`

```
public static void selectionSort(Date[] list) {
  int minPos;
  Date temp;
  for (int index = 0; index < list.length-1; index++)
  {
   minPos = index;
   for (int pos = index+1; pos < list.length; pos++)
       if (list[pos].compareTo(list[minPos]) < 0)
            minPos = pos;
   temp = list[minPos];
   list[minPos] = list[index];
   list[index] = temp;
  }
}
```

Selection Sort Algorithm
on an array of objects that are `Comparable`

```
public static void selectionSort(Comparable[] list) {
  int minPos;
  Comparable temp;
  for (int index = 0; index < list.length-1; index++)
  {
   minPos = index;
   for (int pos = index+1; pos < list.length; pos++)
       if (list[pos].compareTo(list[minPos]) < 0)
            minPos = pos;
   temp = list[minPos];
   list[minPos] = list[index];
   list[index] = temp;
  }
}
```

This is a <u>polymorphic</u> reference.

Binary Search Algorithm

- Start with an array that is already sorted in non-decreasing order.
- Start with the middle value.
- If this is the data value we're looking for (known as the "target"), we're done.
- Otherwise, determine which half of the array the target could be in.
- Find the middle value of that half.
- Repeat this process until we either find the target or we end up with no data values left to search.

Binary Search Algorithm

Searching for 62

18	23	62	81	97	Find middle
18	23	62	81	97	Target found

at position 2

Binary Search Algorithm

Searching for 97

18	23	62	81	97	Find middle
18	23	62	81	97	Not the target
18	23	62	81	97	Find middle
18	23	62	81	97	Not the target
18	23	62	81	97	Find middle
18	23	62	81	97	Target found

at position 4

When you have an even number of data values, choose the value just to the left of the "middle".

Binary Search Algorithm

Searching for 15

18	23	62	81	97	Find middle
18	23	62	81	97	Not the target
18	23	62	81	97	Find middle
18	23	62	81	97	Not the target

Target not found

Binary Search Algorithm

on a sorted array of `int`

```
public static int binarySearch(int[] list, int target)
{
   int min = 0, max = list.length-1, mid = 0;
   boolean found = false;
   while (!found && min <= max) {
     mid = (min + max) / 2;      // (integer division!)
     if (list[mid] == target)
         found = true;
     else if (target < list[mid])
         max = mid-1;
     else       min = mid+1;
   }
   if (found) return mid;
   else return -1;
}
```

Binary Search Algorithm

on a sorted array of `String`

```
public static int binarySearch(String[] list, String target)
{
   int min = 0, max = list.length-1, mid = 0;
   boolean found = false;
   while (!found && min <= max) {
     mid = (min + max) / 2;      // (integer division!)
     if (target.compareTo(list[mid]) == 0)
         found = true;
     else if (target.compareTo(list[mid]) < 0)
         max = mid-1;
     else       min = mid+1;
   }
   if (found) return mid;
   else return -1;
}
```

can also use `target.equals(list[mid])`

Binary Search Algorithm

on a sorted array of `Date`

```
public static int binarySearch(Date[] list, Date target)
{
   int min = 0, max = list.length-1, mid = 0;
   boolean found = false;
   while (!found && min <= max) {
     mid = (min + max) / 2;      // (integer division!)
     if (target.compareTo(list[mid]) == 0)
         found = true;
     else if (target.compareTo(list[mid]) < 0)
         max = mid-1;
     else       min = mid+1;
   }
   if (found) return mid;
   else return -1;
}
```

Binary Search Algorithm

on a sorted array of `Comparable` objects

```
public static int binarySearch(Comparable[] list,
   Comparable target)
{
   int min = 0, max = list.length-1, mid = 0;
   boolean found = false;
   while (!found && min <= max) {
     mid = (min + max) / 2;      // (integer division!)
     if (target.compareTo(list[mid]) == 0)
         found = true;
     else if (target.compareTo(list[mid]) < 0)
         max = mid-1;
     else       min = mid+1;
   }
   if (found) return mid;
   else return -1;
}
```

Summary

- We can use interfaces to specify common behavior amongst various classes.
  - Example: All classes that implement **`Comparable`** must provide a **`compareTo`** method that works in a similar way.
- Interfaces also allow us to write more generic methods that can work on a whole class of objects.
- Polymorphism is an object-oriented principle where a single reference variable can refer to different types of objects at different points in time during the program execution.