Chapter 14

Interfaces

Goals

- Understand what it means to implement a Java interface
- Use the `Comparable` interface to have any type of elements sorted and binary searched
- Show how a Java `interface` can specify a type

14.1 Java Interfaces

Java has 422 interfaces. Of the 1,732 Java classes, 646 classes or 37%, implement one or more interfaces. Considering the large number of interfaces in Java and the high percentage of Java classes that implement the interfaces, interfaces should be considered to be an important part of the Java programming language. Interfaces are used for several reasons:

- Guarantee a class has a particular set of methods and catch errors at compile-time rather than runtime.
- Implement the same behavior with different algorithms and data structures where one may be better in some circumstances, and the other better in other circumstances.

- Treat a variety of types as the same type where the same message results in different behavior
- Provide programming projects that guarantee the required methods have the required method signature
- In larger projects, develop software using an interface before the completed implementation

You will not be asked to write the interfaces themselves (like the `TimeTalker` interface below). Instead, you will be asked to write a class that implements an `interface`. The interface will be given to you in programming projects.

A Java `interface` begins with a heading that is similar to a Java `class` heading except the keyword `interface` is used. A Java `interface` cannot have constructors or instance variables. A Java interface will be implemented by one or more Java classes that add instance variables and have their own constructors. A Java `interface` specifies the method headings that someone decided would represent what all instances of the class must be able to do.

Here is a sample Java interface that has only one method. Although not very useful, it provides a simple first example of an interface and the classes that implement the interface in different ways.

```
// File name: TimeTalker.java
// An interface that will be implemented by several classes
public interface TimeTalker {
  // Return a representation of how each implementing class tells time
  public String tellMeTheTime();
}
```

For each `interface`, there are usually two or more classes that implement it. Here are three classes that implement the `TimeTalker` interface. One instance variable has been added to store the name of any `TimeTalker`. A constructor was also needed to initialize this instance variable with anybody's name.

```java
// File name: FiveYearOld.java
// Represent someone who cannot read time yet.
public class FiveYearOld implements TimeTalker {
  String name;

  public FiveYearOld(String name) {
    name = name;
  }

  public String tellMeTheTime() {
    return name + " says it's morning time";
  }
}
```

```java
// File name: DeeJay.java
// A "hippy dippy" DJ who always mentions the station
public class DeeJay implements TimeTalker {
  String name;

  public DeeJay(String name) {
    name = name;
  }

  public String tellMeTheTime() {
    return name + " says it's 7 oh 7 on your favorite oldies station";
  }
}
```

```
 // File name: FutureOne.java
// A "Star-Trekker" who speaks of star dates
public class FutureOne implements TimeTalker {
  String name;

  public FutureOne(String name) {
    name = name;
  }

  public String tellMeTheTime() {
    return name + " says it's star date 78623.23";
  }
}
```

One of several reasons that Java has interfaces is to allow many classes to be treated as the same type. To demonstrate that this is possible, consider the following code that stores references to three different types of objects as `TimeTalker` variables.

```
  // These three objects can be referenced by variables
  // of the interface type that they implement.
  TimeTalker youngOne = new FiveYearOld("Pumpkin");
  TimeTalker dj = new DeeJay("WolfMan Jack");
  TimeTalker captainKirk = new FutureOne("Jim");
  System.out.println(youngOne.tellMeTheTime());
  System.out.println(dj.tellMeTheTime());
  System.out.println(captainKirk.tellMeTheTime());
```

Output

```
Pumpkin says it's morning time
WolfMan Jack says it's 7 oh 7 on your favorite oldies station
Jim says it's star date 78623.23
```

Because each class implements the `TimeTalker` interface, references to instances of these three classes—`FiveYearOld`, `DeeJay`, and `FutureOne`—can be stored in the reference type variable `TimeTalker`. They can all be considered to be of type `TimeTalker`. However, the same message to the three different classes of `TimeTalker` objects results in three different behaviors. The same message executes three different methods in three different classes.

A Java interface specifies the exact method headings of the classes that need to be implemented. These interfaces capture design decisions—what instances of the class should be able to do—that were made by a team of programmers.

Self-Check

14-1 Write classes `Chicken` and `Cow` that both implement this interface:

```
public interface BarnyardAnimal {
  public String sound();
}
```

The following assertions in this test method must pass.

```
@Test
public void animalsTest() {
  BarnyardAnimal a1 = new Chicken("cluck");
  BarnyardAnimal a2 = new Cow("moo");
  assertEquals("cluck", a1.sound());
  assertEquals("moo", a2.sound());
  a1 = new Chicken("Cluck Cluck");
  a2 = new Cow("Moo Moo");
  assertEquals("Cluck Cluck", a1.sound());
  assertEquals("Moo Moo", a2.sound());
}
```

14.2 The Comparable Interface

The `compareTo` method has been shown to compare two String objects to see if one was less than, greater than, or equal to another. This section introduces a general way to compare any objects with the same `compareTo` message. This is accomplished by having a class implement the `Comparable` interface. Java's `Comparable` interface has just one method—`compareTo`.

```
public interface Comparable<T> {

 /*
  * Returns a negative integer, zero, or a positive integer when this object is
  * less than, equal to, or greater than the specified object, respectively.
  */
  public int compareTo(T other);
}
```

The angle brackets represent a new syntactical element that specifies the type to be compared. Since any class can implement the `Comparable` interface, the `T` in `<T>` will be replaced with the class name that implements the interface. This ensures the objects being compared are the same class. This example shows how one class may implement the `Comparable<T>` interface with as little code as possible (it compiles, but makes no sense):

```
public class AsSmallAsPossible implements Comparable<AsSmallAsPossible> {

  // No instance variables to compare, two AsSmallAsPossible objects are always equal
  public int compareTo(AsSmallAsPossible other) {
    // TODO Auto-generated method stub
    return 0;
  }
}
```

With this incomplete type only shown to demonstrate a class implementing an interface (it is useless otherwise), all `AsSmallAsPossible` objects would be considered equal since `compareTo` always returns 0.

When a class implements the `Comparable` interface, instances of that class are guaranteed to understand the `compareTo` message. Some Java classes already implement `Comparable`.[1] Additionally, any class you write may also implement the `Comparable` interface. For example, to sort or binary search an array of `BankAccount` objects, `BankAccount` can be made to implement `Comparable<BankAccount>`. Then two `BankAccount` objects in an array named `data` can be compared to see if one is less than another:

```
// . . . in the middle of selection sort . . .
// Then compare all of the other elements, looking for the smallest
for(int index = top + 1; index < data.length; index++) {
  if(data[index].compareTo(data[indexOfSmallest]) < 0 )
    indexOfSmallest = index;
}
// . . .
```

The expression highlighted above is true if `data[index]` is "less than" `data[smallestIndex]`. What "less than" means depends upon how the programmer implements the `compareTo` method. The `compareTo` method defines what is known as the "natural ordering" of objects. For strings, it is alphabetic ordering. For `Double` and `Integer` it is what we already know: 3 < 4 and 1.2 > 1.1, for example. In the case of `BankAccount`, we will see that one `BankAccount` can be made to be "less than" another when its ID precedes the other's ID alphabetically. The same code could be used to sort any type of objects as long as the class implements the `Comparable` interface. Once sorted, the same binary search method could be used for any array of objects, as long as the objects implement the comparable interface.

---

[1] Some of the Java classes that implement the `Comparable` interface are `Character`, `File`, `Long`, `ObjectStreamField`, `Short`, `String`, `Float`, `Integer`, `Byte`, `Double`, `BigInteger`, `BigDecimal`, `Date`, and `CollationKey`.

To have a new type fit in with this general method for comparing two objects, first change the class heading so the type implements the `Comparable` interface.

```
public class BankAccount implements Comparable<BankAccount> {
```

The `<T>` in the `Comparable` interface becomes `<BankAccount>`. If the class name were `String`, the heading would use `<String>` as in

```
public class String implements Comparable<String> {
```

Adding **`implements`** is not enough. An attempt to compile the class without adding the `compareTo` method results in this compile time error:

> The type BankAccount must implement the inherited abstract method Comparable<BankAccount>.compareTo(BankAccount)

Adding the `compareTo` method to the `BankAccount` class resolves the compile time error. The heading *must* match the method in the interface. So the method must return an int. And the `compareTo` method *really should* return zero, a negative, or a positive integer to indicate if the object before the dot (the receiver of the message) is equal to, less than, or greater than the object passed as the argument. This desired behavior is indicated in the following test method.

```
@Test public void testCompareTo() {
  BankAccount b1 = new BankAccount("Chris", 100.00);
  BankAccount b2 = new BankAccount("Kim", 100.00);
  // Note: The natural ordering is based on IDs, the balance is ignored
  assertTrue(b1.compareTo(b1) == 0); // "Chris" == "Chris"
  assertTrue(b1.compareTo(b2) < 0);  // "Chris" < "Kim"
  assertTrue(b2.compareTo(b1) > 0);  // "Kim" > "Chris"
}
```

Since the `Comparable` interface was designed to work with any Java class, the `compareTo` method must have a parameter of that same class.

```
/**
  * This method allows for comparison of two BankAccount objects.
  *
  * @param other is the object being compared to this BankAccount.
  * @return a negative integer if this object has an ID that alphabetically
  * precedes other (less than), 0 if the IDs are equals, or a positive
  * integer if this object follows other alphabetically (greater than).
  */
 public int compareTo(BankAccount other) {
   if (this.getID().compareTo(other.getID()) == 0)
     return 0;  // This object "equals" other
   else if (this.getID().compareTo(other.getID()) < 0)
     return -1; // This object < other
   else
     return +1; // This object > other
 }
```

Or since String's `compareTo` method already exists, this particular `compareTo` method can be written more simply.

```
 public int compareTo(BankAccount other) {
   return this.getID().compareTo(other.getID());
 }
```

The Implicit Parameter `this`

The code shown above has `getID()` messages sent to **`this`**. In Java, the keyword **`this`** is a reference variable that allows an object to refer to itself. When **`this`** is the receiver of a message, the object is using its own internal state (instances variables). Because an object sends messages to itself so frequently, Java provides a

shortcut: **this** and the dot are not really necessary before the method name. Whereas the keyword **this** is sometimes required, it was not really necessary in the code above. It was used only to distinguish the two objects being compared. Therefore the method could also be written as follows:

```
public int compareTo(BankAccount other) {
  return getID().compareTo(other.getID()); // "this." removed
}
```

It's often a matter of taste of when to use **this**. You rarely need **this**, but **this** sometimes clarifies things. Here is an example where **this** is needed. Since the constructor parameters have the same names as the instance variables, the assignments currently have no effect.

```
public class BankAccount implements Comparable<BankAccount> {

  // Instance variables that every BankAccount object will maintain.
  private String ID;
  private double balance;

  public BankAccount(String ID, double balance) {
    ID = ID;
    balance = balance;
  }
```

If you really want to name instance variables the same as the constructor parameters, you must write **this** to distinguish the two. With the code above, the instance variables will never change to the expected values of the arguments. To fix this error, add **this** to distinguish instance variables from parameters.

```
  public BankAccount(String ID, double balance) {
    this.ID = ID;
    this.balance = balance;
  }
```

Self-Check

14-2 Modify `compareTo` so it defines the natural ordering of `BankAccount` based on balances rather than IDs. One account is less than another if the balance is less than the other. The following assertions must pass.

```
@Test
public void testCompareTo() {
  BankAccount b1 = new BankAccount("Chris", 111.11);
  BankAccount b2 = new BankAccount("Chris", 222.22);
  // Note: The natural ordering is based on the balance. IDs are ignored.
  assertTrue(b1.compareTo(b1) == 0); // 111.11 == 111.11
  assertTrue(b1.compareTo(b2) < 0);  // 111.11  < 222.22
  assertTrue(b2.compareTo(b1) > 0);  // 222.22 > 111.11
}
```

14.3 New Types Specified as Java Interfaces

The Java `interface` can also be used to specify a type. For example, the following Java interface specifies the operations for a String-like type that has methods that actually change the objects of any class that implements `MutableString`.

```
public interface MutableString {

  /**
   * Return the number of characters in this object
   */
  public int length();
```

```
  /**
   * Returns the character in this sequence at the specified index
   */
  public char charAt(int index);

  /**
   * Change all lower case letters to upper case.
   */
  public void toUpperCase();

  /**
   * Replaces each occurrence of oldChar with newChar. If oldChar
   * does not exist, no change is made to this object
   */
  public void replace(char oldChar, char newChar);

  /**
   * Add the chars in array at the end of this object.
   */
  public void concatenate(char [] array);
}
```

An `interface` does not specify instance variables, constructors, or the algorithms for the methods specified in the interface. Comments and well-named identifiers imply the behavior of operations. This behavior can be made much more explicit with assertions. For example, the assertions shown in the following test methods help describe the behavior of `add` and size. This code assumes that a class named `OurString` implements `interface MutableString` and a constructor exists that takes a n array of `char` to initialize `OurString` objects.

```
@Test
public void testGetters() {
  char[] toAdd = { 'a', 'b', 'c' };
  MutableString s = new OurString(toAdd);
  assertEquals(3, s.length());
  assertEquals('a', s.charAt(0));
  assertEquals('b', s.charAt(1));
  assertEquals('c', s.charAt(2));
}

@Test
public void testMakeUpper() {
  MutableString s = new OurString(new char[] { 'a', '&', '1', 'z' });
  s.toUpperCase();
  assertEquals('A', s.charAt(0));
  assertEquals('&', s.charAt(1));
  assertEquals('1', s.charAt(2));
  assertEquals('Z', s.charAt(3));
}

@Test
public void testConcatenate() {
  char[] a1 = { 'a', 'b', 'c' };
  MutableString s = new OurString(a1);
  // Pass an new array of char to concatenate as an argument
  s.concatenate(new char[] { ',', 'D' });
  assertEquals(5, s.length());
  assertEquals('a', s.charAt(0));
  assertEquals('b', s.charAt(1));
  assertEquals('c', s.charAt(2));
  assertEquals(',', s.charAt(3));
  assertEquals('D', s.charAt(4));
}
```

Since the interface does not specify constructors and instance variables, the programmer is left to design the name of the class, the constructor, and a way to store the state of the objects. In the following design, the constructor takes an array of char and stores the characters in the first `array.length` locations of the `char[]` instance variable. Notice that the array is bigger than need be. This design uses a buffer--a place to store new characters during concatenate without growing the array.

```
public class OurString implements MutableString {

  private char[] theChars;
  private int n; // the number of meaningful characters in this object

  /**
   * Construct a mutable OurString object with an array of characters
   */
  public OurString(char[] array) {
    n = array.length;
    theChars = new char[128];
    for (int i = 0; i < n; i++)
      theChars[i] = array[i];
  }

  /**
   * Return the number of chars in this OurString object
   */
  public int length() {
    return n;
  }
```

```
  /**
   * Returns the character in this sequence at the specified index. The first
   * char value is at index 0, the next at index 1, and so on, as in array
   * indexing. The index argument must be greater than or equal to 0, and less
   * than the length of this sequence of characters
   */
  public char charAt(int index) {
    return theChars[index];
  }

  // The other methods are written as stubs that need to be implemented.
  // They don't work, but they are needed for this class to compile.
  public void concatenate(char[] array) {
    // TODO Auto-generated method stub
  }

  public void replace(char oldChar, char newChar) {
    // TODO Auto-generated method stub
  }

  public void toUpperCase() {
    // TODO Auto-generated method stub
  }
}
```

Note: If you are using an integrated development environment (IDE), you can quickly obtain a class that implements an interface. That class will have all methods from the interface written as stubs to make things compile. A stub has a method heading and a body. For non void functions, the IDE will add some default return values such as `return 0;` from an `int` method.

Completing the other three methods in `OurString` is left as an optional exercise.

Answers to Self-Check Questions

14-1
```
public class Chicken implements BarnyardAnimal {
  private String mySound;

  public Chicken(String sound) {
    mySound = sound;
  }

  public String sound() {
    return mySound;
  }
}

public class Cow implements BarnyardAnimal {
   private String mySound;

   public Cow(String sound) {
     mySound = sound;
   }

   public String sound() {
     return mySound;
   }
 }
```

14-2
```
public int compareTo(BankAccount other) {
  double thisObjectsPennies = 100 * this.getBalance();
  double theOtherObjectsPennies = 100 * other.getBalance();
  return (int) thisObjectsPennies - (int) theOtherObjectsPennies;
}
```