14.3 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.