Code from Java API classes

To give you a better understanding of how the classes `String`, `StringBuilder`, and `ArrayList` work, I'll explain the variables used to store these objects' values, along with definitions for some of their methods. My purpose is not to overwhelm you but to prepare you. The exam won't question you on this subject, but these details will help you retain relevant information for the exam and implement similar requirements in code for practical projects.

The source code of the classes defined in the Java API is shipped with the Java Development Kit (JDK). You can access it by unzipping the src.zip archive from your JDK's installation folder.

The rest of this section discusses how the authors of the Java API have implemented immutability in the class `String`.

STRING USES A CHAR ARRAY TO STORE ITS VALUE

Here's a partial definition of the class `String` from the Java source code file (String.java) that includes the array used to store the characters of a `String` value (the relevant code is in bold):

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence
{
    private final char value[];

}
```

The value array is used for character storage.

The rest of the code of the class String

The arrays are fixed in size—they can't grow once they're initialized.

Let's create a variable `name` of type `String` and see how it's stored internally:

```
String name = "Selvan";
```

Figure 4.5 shows a UML representation (class diagram on the left and object diagram on the right) of the class `String` and its object `name`, with only one relevant variable, `value`, which is an array of the type `char` and is used to store the sequence of characters assigned to a `String`.

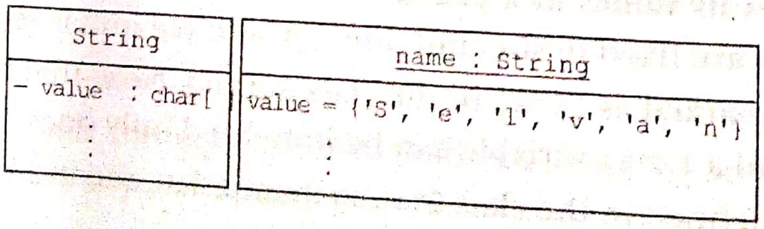

Figure 4.5 UML representations of the class `String` and a `String` object with `String`'s instance attribute `value`

As you can see in figure 4.5, the `String` value `Selvan` is stored in an array of type `char`. In this chapter, I'll cover arrays in detail, as well as how an array stores its first value at position 0.

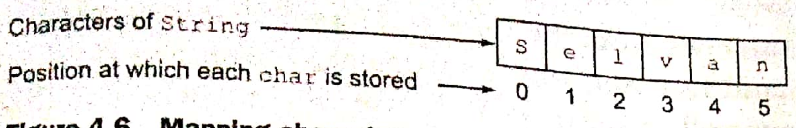

Figure 4.6 Mapping characters stored by a `String` with the positions at which they're stored

Figure 4.6 shows how `Selvan` is stored as a `char` array.

What do you think you'll get when you request that this `String` return the character at position 4? If you said `a` and not `v`, you got the right answer (as in figure 4.6).

STRING USES FINAL VARIABLE TO STORE ITS VALUE

The variable `value`, which is used to store the value of a `String` object, is marked as `final`. Review the following code snippet from the class `String.java`:

```
private final char value[];
```

value is used for character storage.

The basic characteristic of a `final` variable is that it can initialize a value only once. By marking the variable `value` as `final`, the class `String` makes sure that it can't be reassigned a value.

METHODS OF STRING DON'T MODIFY THE CHAR ARRAY

Although we can't reassign a value to a `final` `char` array (as mentioned in the previous section), we can reassign its individual characters. Wow—does this mean that the statement "`Strings` are immutable" isn't completely true?

No, that statement is still true. The `char` array used by the class `String` is marked `private`, which means that it isn't accessible outside the class for modification. The class `String` itself doesn't modify the value of this variable either.

All the methods defined in the class `String`, such as `substring`, `concat`, `toLowerCase`, `toUpperCase`, `trim`, and so on, which *seem* to modify the contents of the `String` object on which they're called, create and return a new `String` object rather than modify the existing value. Figure 4.7 illustrates the partial definition of `String`'s `replace` method.

```
public String replace(char oldChar, char newChar) {
    if (oldChar != newChar) {
        // code to create a new char array and
        // replace the desired char with the new char

        return new String(0, len, buf);
    }
    return this;
}
```

`replace` creates and returns a new `String` object. It doesn't modify the existing array value.

Figure 4.7 The partial definition of the method `replace` from the class `String` shows that this method creates and returns a new `String` object rather than modifies the value of the `String` object on which it's called.