

# Chapter 16

## Strings

---

### Java AP

**Copyright Notice**

Copyright © 2013 DigiPen (USA) Corp. and its owners. All Rights Reserved

No parts of this publication may be copied or distributed, transmitted, transcribed, stored in a retrieval system, or translated into any human or computer language without the express written permission of DigiPen (USA) Corp., 9931 Willows Road NE, Redmond, WA 98052

**Trademarks**

DigiPen® is a registered trademark of DigiPen (USA) Corp.

All other product names mentioned in this booklet are trademarks or registered trademarks of their respective companies and are hereby acknowledged.

## Strings

- String is a class in Java that allows us to create objects that consist of sequences of characters.
- The String class also contains multiple methods that allow us to manipulate the sequence of characters stored inside a string.
- We have been using strings since the beginning of this course but so far all of them have been string literals.

Example	<pre>public class Main {     public static void main(String[] args)     {         System.out.println("I love java!");     } }</pre>
Explanation	<p>"I love java" is a string literal that we output to the console. When the compiler encounters a string literal, it creates a String object for you. In this chapter, we will learn how to create String objects ourselves, fill them with sequences of characters, and manipulate them using code.</p> <p><b><u>Note:</u></b> Any sequence of characters between quotes (" ") creates a string literal.</p>

### Creating a String

- Like any class instance, a **String** instance is created using the **new** keyword followed by the **String class' constructor**.
- With the **String** class, it is also possible to create an instance by assigning a string literal to a String variable. The compiler will create a String object out of the string literal and assign it to the instance.

Example	<pre>public class Main {     public static void main(String[] args)     {         String s1 = new String("Used the String constructor");         System.out.println(s1);          String s2 = "Used a string literal";         System.out.println(s2);     } }</pre>
---------	--

Output	Used the String constructor Used a string literal
--------	--

### String concatenation

- Concatenation allows us to combine multiple strings into one.
- Strings are concatenated using the "+" operator.

Example	<pre> public class Main {     public static void main(String[] args)     {         String s1 = "We " + "can combine " + "strings";         System.out.println(s1);          String s2 = "Hello";         String s3 = "World";         String s4 = s2 + " " + s3;         System.out.println(s2);         System.out.println(s3);         System.out.println(s4);          int num = 5;         String s5 = "We can concatenate numbers: " + num;         System.out.println(s5);     } } </pre>
Output	<pre> We can combine strings Hello World Hello World We can concatenate numbers: 5 </pre>
Explanation	<p>First, three string literals are concatenated into one string object (<b>s1</b>). Then string objects (<b>s2</b> and <b>s3</b>) and string literals (" ") are concatenated into another string object (<b>s4</b>).</p> <p>Lastly, a string literal and an integer are concatenated into a string object (<b>s5</b>). In this last part, the compiler is creating a string object out of the integer and then concatenating it with the other string object. We will touch more on this at the end of the chapter.</p>

## ASCII and char

- As previously mentioned, a string in Java consists of a sequence of characters.
- Characters can be represented in a variety of ways, but the one we will be using is ASCII.
- ASCII allows us access to letters (upper case and lower case), numbers and punctuation that we need for the English language (and languages that use the same set of characters).
- The machine treats ASCII characters as integers between 0 and 255. In other words, a string is really a sequence of integers with values ranging from 0 to 255.
- characters (**char**) are initialized with a character literal, denoted by a pair of single quotes (' ').

### Example

```
char character = 'A';
```

- A character can be treated as a number, allowing us to modify the value with simple arithmetic operations.

### Example

```
public class Main
{
    public static void main(String[] args)
    {
        char character = 'A';
        int characterValue = character;
        System.out.println("character: " + character);
        System.out.println("character value: " + characterValue);

        character = 'A' + 3;
        characterValue = character;
        System.out.println("character: " + character);
        System.out.println("character value: " + characterValue);

        character = 'a' - 32;
        characterValue = character;
        System.out.println("character: " + character);
        System.out.println("character value: " + characterValue);
    }
}
```

### Output

```
character: A
character value: 65
character: D
character value: 68
character: A
character value: 65
```

**Explanation**

Characters are treated as numbers. Adding 3 to an 'A' is exactly like adding 3 to 65.

Note: the char type can't hold a string. The following code:

```
char character = "Hello";
```

will lead to a compiler error:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
```

```
    Type mismatch: cannot convert from String to char
```

```
    at Main.main(Main.java:5)
```

**Most Used ASCII Table Values**

Range	First Character in Range	Last Character in Range
48 - 57	0	9
65 - 90	A	Z
97 - 122	a	z

**Note:** If you want to find the other values a simple search on the internet will lead to numerous complete tables.

## String Methods

- The String class provides many methods that help us manipulate string objects. We will be covering a few of them in this chapter, but feel free to experiment with the rest on your own.

### charAt

Method Signature	Description
<b><i>char charAt(int index)</i></b>	Returns the character value at a specified index in the String, 0 being the first character.

Example	<pre> public class Main {     public static void main(String[] args)     {         String s = "Coding is awesome!";          char character = s.charAt(0);         System.out.println("character at index 0: " + character);          character = s.charAt(5);         System.out.println("character at index 5: " + character);     } } </pre>
Output	<pre> character at index 0: C character at index 5: g </pre>
Explanation	<p>As mentioned before, a string is a sequence of characters. The charAt method returns the character at a specific index in the string.</p> <p><b>Note:</b></p> <ul style="list-style-type: none"> <li>Like most things in programming, a string is 0 based, which means the index of its first character is 0.</li> <li>Attempting to access an index outside of the array will lead to a compiler error. Example: <code>character = s.charAt(100);</code></li> </ul> <pre> Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: 100     at java.lang.String.charAt(Unknown Source)     at Main.main(Main.java:13) </pre>

### length

Method Signature	Description
<b><i>int length()</i></b>	Returns the length of a string object (the number of characters in the string).

Example	<pre> public class Main {     public static void main(String[] args)     {         String s = "Hello";         System.out.println(s.length());     } } </pre>
Output	5

### ***toLowerCase() & toUpperCase()***

Method Signature	Description
<b><i>String toLowerCase()</i></b>	Returns a new String where every upper case character has been replaced by its lower case equivalent.
<b><i>String toUpperCase()</i></b>	Returns a new String where every lower case character has been replaced by its upper case equivalent.

Example	<pre> public class Main {     public static void main(String[] args)     {         String mixed = "AbCdEfGhI!?" ;         String upper = mixed.toUpperCase();         String lower = mixed.toLowerCase();         System.out.println("Mixed: " + mixed);         System.out.println("Upper: " + upper);         System.out.println("Lower: " + lower);     } } </pre>
---------	---

<b>Output</b>	Mixed: AbCdEfGhI!? Upper: ABCDEFGHI!? Lower: abcdefghi!?
<b>Explanation</b>	<p>The <b>toUpperCase</b> and <b>toLowerCase</b> methods only change letters to the requested casing. All non letters characters remain the same.</p> <p><b>Note:</b> <i>Neither method changes the letters in the original string. A new String object is returned that contains the changed content.</i></p>

### indexOf

Method Signature	Description
<b><i>int indexOf(String str)</i></b>	Returns the index of the first occurrence of a given string ( <b>str</b> ) in the calling String object. A <b>-1</b> value is returned if no occurrence is found.
<b><i>int indexOf(String str, int fromIndex)</i></b>	Returns the index of the first occurrence of a given string ( <b>str</b> ) in the calling String object starting at a specified location in the string ( <b>fromIndex</b> ). A <b>-1</b> value is returned if no occurrence is found.

<b>Example</b>	<pre> public class Main {     public static void main(String[] args)     {         String s = "I love coding";          int index = s.indexOf("coding");         System.out.println(index);          index = s.indexOf("Coding");         System.out.println(index);          String s2 = "ILoveCoding";         index = s2.indexOf("Coding");         System.out.println(index);          String s3 = "Even coding loves coding";         index = s3.indexOf("coding");         System.out.println(index);     } } </pre>
----------------	--



	<pre>         index = s3.indexOf("coding", 6);         System.out.println(index);     } } </pre>
Output	<pre> 7 -1 5 5 18 </pre>
Explanation	<p>The <b>indexOf</b> method finds the first occurrence of a user specified string inside another string and returns the index of its first character.</p> <p><b>indexOf</b> starts the search at the beginning of the string unless a starting index is specified by the user.</p> <p><b>Note:</b> This method is case sensitive, so trying to find the first occurrence of "coding" is completely different than trying to find "Coding".</p>

### substring

Method Signature	Description
<b><i>String substring(int begin)</i></b>	Returns a new String object containing the characters from the <b>begin</b> index until the end of the calling String object.
<b><i>String substring(int begin, int end)</i></b>	Returns a new String object containing the characters from the <b>begin</b> index until the <b>end</b> Index (not including the end index).

Example	<pre> public class Main {     public static void main(String[] args)     {         String s = "I love coding";          String s2 = s.substring(2);         System.out.println(s2);          String s3 = s.substring(2, 6);         System.out.println(s3);     } } </pre>
---------	--

<b>Output</b>	love coding love
---------------	---------------------

<b>Example 2</b>	<pre> public class Main {     public static void main(String[] args)     {         String s = "I love coding";          String s2 = s.substring(-1);         System.out.println(s2);     } } </pre>
<b>Output</b>	<p>Exception in thread "main" <a href="#">java.lang.StringIndexOutOfBoundsException</a>: String index out of range: -1 at java.lang.String.substring(Unknown Source) at Main.main(Main.java:7)</p>
<b>Example 3</b>	<pre> public class Main {     public static void main(String[] args)     {         String s = "I love coding";          String s2 = s.substring(5, 50);         System.out.println(s2);     } } </pre>
<b>Output</b>	<p>Exception in thread "main" <a href="#">java.lang.StringIndexOutOfBoundsException</a>: String index out of range: 50 at java.lang.String.substring(Unknown Source) at Main.main(Main.java:7)</p>
<b>Explanation</b>	<p>These examples prove that accessing indices outside the string's range will lead to a compiler error.</p>

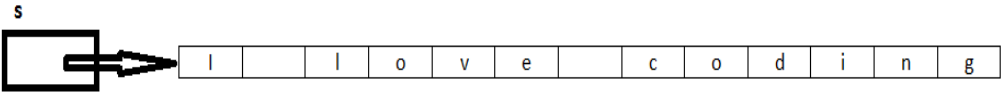
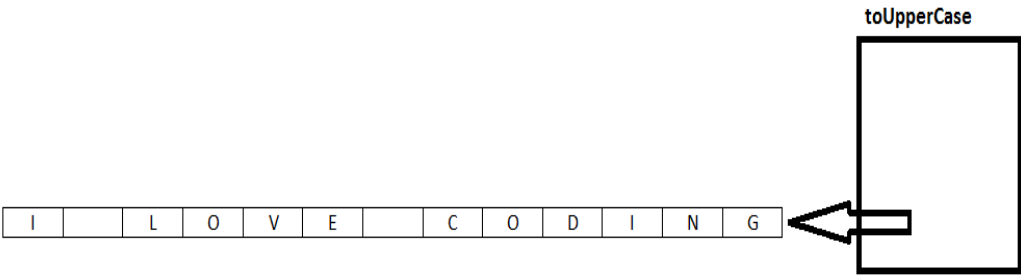
**replace**

Method Signature	Description
<b><i>String replace(char oldChar, char newChar)</i></b>	Replaces all instances of the <b>oldChar</b> with the <b>newChar</b> . A new String object containing the result is returned, the calling String stays intact.
<b><i>String replace(CharSequence target, CharSequence replacement)</i></b>	Replaces all instances of the <b>target</b> CharSequence with the <b>replacement</b> CharSequence. A new String object containing the result is returned, the calling String stays intact. <b><i>Note: A CharSequence is simply a String in this context.</i></b>

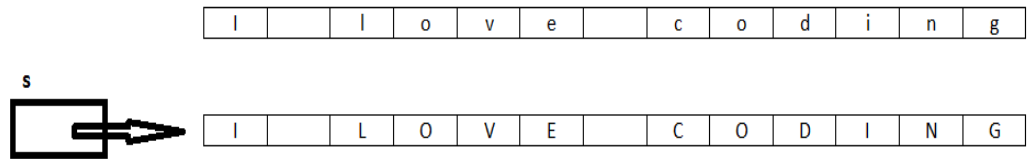
<b>Example</b>	<pre> public class Main {     public static void main(String[] args)     {         String s = "I love coding";          String s2 = s.replace('o', '0');         System.out.println(s2);          String s3 = s.replace("love", "like");         System.out.println(s3);     } } </pre>
<b>Output</b>	<pre> I 10ve c0ding I like coding </pre>

## String Mutability

- String objects are **immutable**. This means they cannot be modified after creation.
- For that reason, all String methods return the resulting string as a new object instead of changing the calling string. For example, when calling **toUpperCase** on a string object, the method will return a new string object with all uppercase characters instead of changing the characters in the original string.
- There are, however, ways to change the original string variable. See the example below:

Example	<pre> public class Main {     public static void main(String[] args)     {         String s = "I love coding";         s = s.toUpperCase();         System.out.println(s);     } } </pre>
Output	I LOVE CODING
Explanation	<p>By reassigning the <b>toUpperCase</b> method's returned string to the caller string, the original string is now replaced by its uppercase version.</p> <p>String s = "I love coding";</p>  <p>s.toUpperCase();</p> 

```
s = s.toUpperCase();
```



The String variable now points to a different String object, specifically the one created using the **toUpperCase** method.

## Comparing Strings

### *equals* method

<b>Example</b>	<pre>public class Main {     public static void main(String[] args)     {         String s1 = new String("Hello");         String s2 = new String("Hello");          System.out.println("s1 == s2 =&gt; " + (s1 == s2));         System.out.println("s1.equals(s2) =&gt; " + s1.equals(s2));     } }</pre>
<b>Output</b>	<pre>s1 == s2 =&gt; false s1.equals(s2) =&gt; true</pre>
<b>Explanation</b>	<p>String objects are references, when using the <code>==</code> operator to check if two strings are equal their references are being compared and not their character sequence.</p> <p>In order to check if two strings contain the same character sequence, the <b>equals</b> method should be used. The <b>equals</b> method returns <b>true</b> if the strings are equal and <b>false</b> if they are not.</p>

**compareTo**

Method Signature	Description
<b><i>int compareTo(String anotherString)</i></b>	<p>Compares two strings.</p> <p>Returns <b>0</b> if the two strings are equal.</p> <p>Returns a <b>negative</b> integer if the calling string alphabetically precedes the string that was passed in (<b>anotherString</b>).</p> <p>Returns a <b>positive</b> number if the string that was passed in (<b>anotherString</b>) alphabetically precedes the calling string.</p>

Example	<pre> public class Main {     public static void main(String[] args)     {         String s1 = new String("Hello");         String s2 = new String("Hello");         String s3 = new String("Hello World");         String s4 = new String("He");         String s5 = new String("homework");          System.out.println(s1.compareTo(s2));         System.out.println(s1.compareTo(s3));         System.out.println(s1.compareTo(s4));         System.out.println(s1.compareTo(s5));     } } </pre>
Output	<pre> 0 -6 3 -32 </pre>
Explanation	<p><u>Comparing s1 and s2:</u> Returned value is 0 because both strings are equal.</p> <p><u>Comparing s1 and s3:</u> Returned value is -6 because s3 contains 6 more characters than s1. In other words, <math>s1.length() - s3.length() = -6</math>. This case happens when the passed in string contains the same sequence as the calling string with some extra characters at the end.</p>

Comparing s1 and s4: Returned value is 3 because s1 contains 3 more characters than s4. In other words, `s1.length() - s4.length() = 3`. This case happens when the passed in string contains the same sequence as the calling string with some missing characters at the end.

Comparing s1 and s5: Returned value is -32, which is the difference in ASCII values between the first two characters in the Strings that are not equal. In this case the first letter in both string are different 'H' and 'h'. 'H' - 'h' = -32.

## Object Class

- All classes in java extend from the same superclass: the **Object** class. This means that every class in Java starts with a set of default methods inherited from the Object class.
- One of these methods is the **equals()** method which was shown in above examples – it helped us compare two strings by value.
- Another important method of the Object class is the **toString()** method, which is called anytime we need a representation of an object as a string of characters.

Method Signature	Description
<b><i>bool equals(Object obj)</i></b>	Compares the passed in object ( <b>obj</b> ) to the calling object.
<b><i>String toString()</i></b>	Returns a string that represents the contents of the class. Every class will override this method and will decide how to form the returned string.

- Assume we have the following Vector class:

<b>Vector3</b>	<pre> public class Vector3 {     public float x,y,z;      public Vector3(float x_, float y_, float z_)     {         x = x_;         y = y_;         z = z_;     } } </pre>
----------------	---

- As in the previous examples with comparing strings, comparing two Vector3 instances with the == operator will just compare their references. Checking if two objects are equal is done only by using the **equals** method. Since every class extends from the Object class, every class can override the inherited **equals** methods to specify how to check the equality of two instances of a class.

Vector3	<pre> public class Vector3 {     public float x,y,z;      public Vector3(float x_, float y_, float z_)     {         x = x_;         y = y_;         z = z_;     }      public boolean equals(Object obj)     {         if(obj instanceof Vector3)         {             Vector3 other = (Vector3)obj;             if(x == other.x &amp;&amp; y == other.y &amp;&amp; z == other.z)             {                 return true;             }         }         return false;     } } </pre>
Explanation	<p>Now that the <b>Vector3</b> class defined the <b>equals</b> method's behavior we can use that instead of the == operator to check if two vectors are equal.</p> <p><b>Note:</b> We are using the "instanceof" operator to make sure that the passed in object is a Vector3 object otherwise we should not compare values and just return false.</p>

Example	<pre> public class Main {     public static void main(String[] args)     {         Vector3 v1 = new Vector3(1.0f, 2.0f, 3.0f);         Vector3 v2 = new Vector3(1.0f, 2.0f, 3.0f);          System.out.println("v1 == v2 =&gt; " + (v1 == v2));         System.out.println("v1.equals(v2) =&gt; " + v1.equals(v2));     } } </pre>
---------	--



Output	<pre>v1 == v2 =&gt; false v1.equals(v2) =&gt; true</pre>
--------	--

## toString

- The **toString** method is called anytime a class object is needed as a String. Believe it or not, that happens a lot. Every time we print an integer to the Console, that integer is changed to a String first then printed to the console.
- Let's try to print an instance of our Vector3 class to the console and see what happens:

Example	<pre>public class Main {     public static void main(String[] args)     {         Vector3 v = new Vector3(1.0f, 2.0f, 3.0f);         System.out.println(v);     } }</pre>
Output	Vector3@689ba632

- What we got is the content of the reference, which is a memory location that the reference is pointing at. Of course what we wanted is the values of x, y and z.
- This is where the **toString** method comes to play. By overriding the **toString** method the class specifies the **String** that should be returned when **println** attempts to print the class instance to the console.

Example	<pre>public class Vector3 {     public float x,y,z;      public Vector3(float x_, float y_, float z_)     {         x = x_;         y = y_;         z = z_;     }      public String toString()     {         return "[" + x + ", " + y + ", " + z + " ]";     } }</pre>
---------	--

	<pre>public class Main {     public static void main(String[] args)     {         Vector3 v = new Vector3(1.0f, 2.0f, 3.0f);         System.out.println(v);     } }</pre>
Output	[ 1.0, 2.0, 3.0 ]