The **String** types

eazy bytes

Strings can be created using string **literals** or by using the **new operator**: You can create a string by enclosing a sequence of characters in double quotes, like this: ***String str = "hello";*** or you can also create a string using the new operator, like this: ***String str = new String("hello");***

Lets see what is the difference between them

When we create a String using the literals like shown below, the String will be stored inside a space called **String pool**

```
String str = "hello";
```

String pool is a special area of heap memory that stores a pool of unique string literals in Java. When you create a string literal in your Java code, the JVM checks if the string already exists in the string pool. If it does, then the existing string is returned from the pool instead of creating a new string object. This is called string interning.

When we create a String using the new operator like shown below, the String will be stored inside the **heap memory**

```
String str = new String("hello");
```

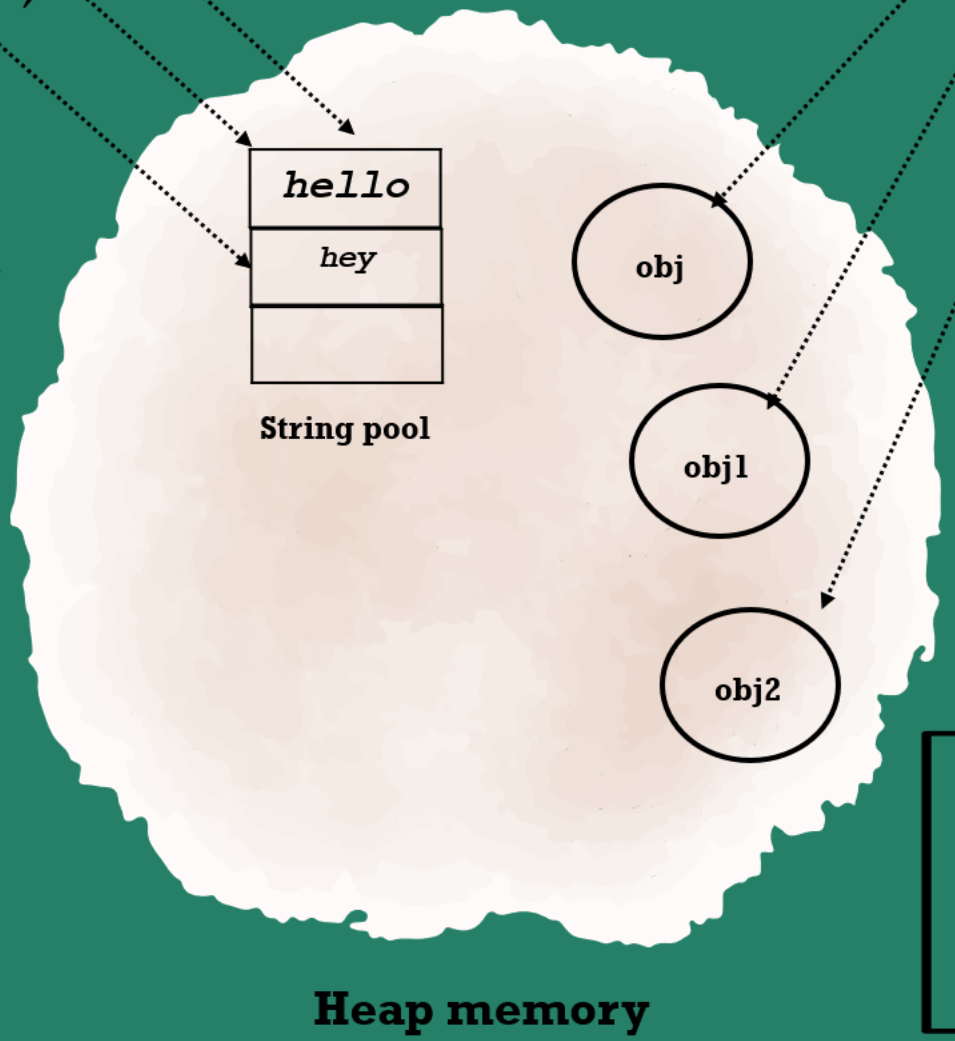
If we create a String object using the new operator, a new object will be created in memory every time, even if the string value is the same. In contrast, if the string is created using literals, the same memory location is used in the String pool.

The **String** types

eazy bytes

```
String hello = "hello";
String hi = "hello";
String hey = "hey";
```

```
String obj = new String("hello");
String obj1 = new String("hello");
String obj2 = new String("hey");
```




```
System.out.println(hello == hi);
```

Using ==, we can compare 2 objects if they are equal & stored in the same memory location. Here, since JVM uses String pool space, the output will be **true**

```
System.out.println(obj == obj1);
```

Using ==, we can compare 2 objects if they are equal & stored in the same memory location. Here since we used new operator, JVM uses heap space & the output will be **false**

> Direct access to the String pool is not possible, and there is no mechanism to remove String objects from the pool, except by restarting the application.

Which String declaration should I use ?

eazy bytes

It is always recommended to use literal style of declaration instead of new operator. Why ? Let's see....

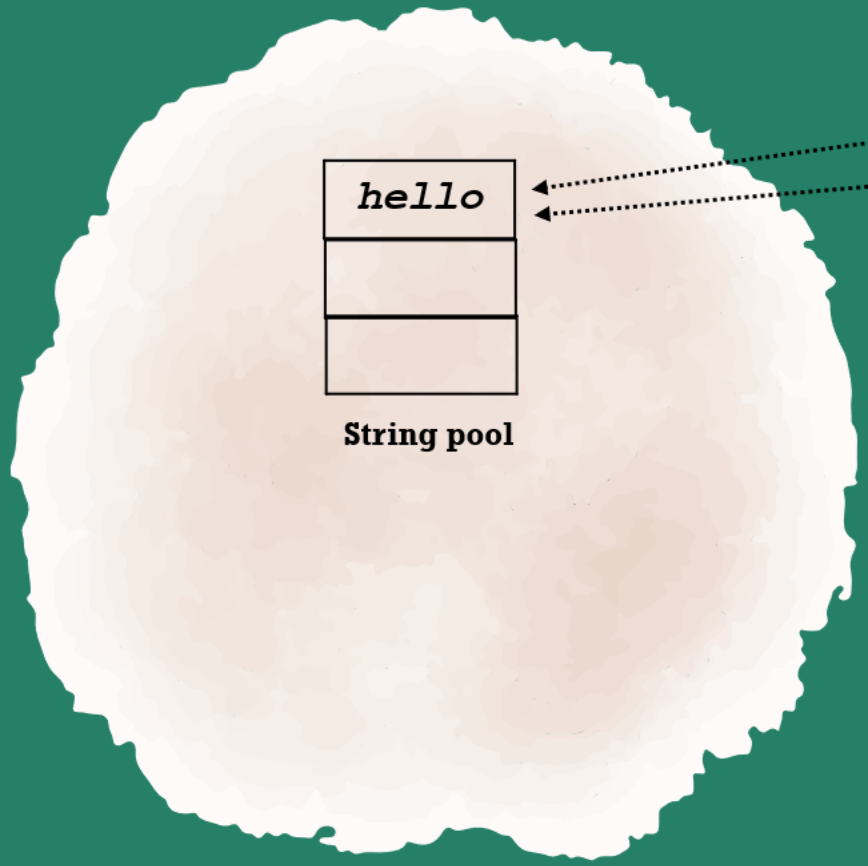
> The main reason for having a string pool in Java is to save memory. Since strings are immutable, meaning their values cannot be changed once they are created, it makes sense to reuse the same string object whenever possible instead of creating new ones. By doing this, the JVM can avoid creating unnecessary string objects and thus reduce memory usage.

> Another advantage of the string pool is that it can improve performance. Since string interning allows the JVM to reuse existing string objects, it can reduce the time and resources needed to create new strings. This can result in faster code execution and improved overall performance. In summary, string pools in Java are used to save memory and improve performance by reusing existing string objects whenever possible.

The **intern** method in String

eazy bytes

If necessary, it is possible to move the string value created with the new operator to the string pool using the intern() method, like below. The intern() method attempts to move the newly created "hello" value into the string pool but discovered that such a literal exists there already, so it reused the literal from the string pool. That is why the print out statement will print the output as true




```
String hello = "hello";
String obj = new String("hello").intern();
System.out.println(hello==obj);
```

Another example

```
String s1 = "hello";
String s2 = new String("hello");
System.out.println(s1 == s2);  // false
String s3 = s2.intern();
System.out.println(s1 == s3);  // true
```

The **concat** method in String

eazy bytes

A single String literal can be formed by combining multiple String literals. You can concatenate two or more strings using the + operator, like show below. No other arithmetic operator can be applied to a String literal or an object

We can combine two strings using **concat()** method as well

```
String hello = "Hello"+ " "+"World"; // Hello World
String concatString = "Hello".concat(" ").concat("World"); // Hello World
```

In Java, an empty string ("") and a null string (null) are not the same.

An empty string is a valid instance of a String object that contains zero characters.
It is essentially a string with a length of 0.

On the other hand, a null string means that the reference variable does not point to any object in the memory. It indicates the absence of a value or an uninitialized state. If a variable of type String is assigned the value null, it means that it does not refer to any string object.

```
// Both are not same
String emptyString = "";
String nullString = null;
```

Escape sequence character & Unicode char values in **String**

eazy bytes

Some times, you may want to store a String value along with double quotes. In such scenarios, using " directly inside your String value will result in compilation error. To overcome this challenge, we can use escape sequence character \" like shown below,

```
String name = "\"Madan\"";
System.out.println(name); // "Madan"
```

We can use Unicode char values inside the String. Behind the scenes, Java compiler will convert the Unicode char value to a normal char and use the same inside String data. For example like shown below, I can use \u004D & \u0021 which are the unicode values of M and !

```
String name = "\u004Dadan\u0021";
System.out.println(name); // Madan!
```

Finding the **length** of a String

eazy bytes

In Java, the length() method is a built-in method of the String class, and it is used to find the length or the number of characters in a given string. The return type of length() method is int representing the number of characters in the string.

The length is determined by counting the characters in the string, including spaces and special characters. If the string is empty, length() will return 0.

```
String str1 = "Hello";
String str2 = "Java";
String combined = str1 + " " + str2;

int length1 = str1.length(); // length1 is 5
int length2 = str2.length(); // length2 is 4
int combinedLength = combined.length(); // combinedLength is 10
```

In Java, when you use a string literal (like "Hello"), the compiler automatically creates a corresponding String object. This means that for practical purposes, a string literal is treated the same as a String object. For instance, if you have the statement String str1 = "Hello";, the compiler creates a String object with the content "Hello".

All methods of the String class are applicable directly to string literals. For example, you can calculate the length of a string literal using the length() method:

```
int len1 = "".length(); // len1 is 0
int len2 = "Hello".length(); // len2 is 5
```

Comparing Strings

eazy bytes

When working with strings in Java, you might need to compare them to check if they are the same or to sort them in a specific order. Here's how you can do that:

Comparing for Equality:

To check if two strings have the same content, you use the equals() method provided by the String class. For example:

```
String str1 = new String("Hey");
String str2 = new String("Hello");
String str3 = new String("Hey");

boolean b1 = str1.equals(str2);  // This will be false
boolean b2 = str1.equals(str3);  // This will be true
```

You can also compare string literals or mix string literals with String objects.

> Remember that using the == operator compares references in memory, not the actual content

```
String str1 = new String("Hey");
String str2 = new String("Hello");
String str3 = new String("Hey");

b1 = str1.equals("Hey"); // This will be true
b2 = "Hey".equals(str1); // This will be true
b1 = "Hey".equals("Hello");// This will be false
```

Comparing Strings

eazy bytes

If you want to compare two strings and not care about whether the letters are uppercase or lowercase, you can use the **equalsIgnoreCase()** method. On the other hand, if you want to consider the case (uppercase and lowercase letters) in the comparison, you should use the **equals()** method.

```
String str1 = "java";
String str2 = "JAVA";
boolean isSame = str1.equalsIgnoreCase(str2); // true
```

If you want to compare strings for sorting purposes based on Unicode values, you can use the compareTo() method of the String class. It returns an integer.

```
int comparison1 = "java".compareTo("java"); // This will be 0
int comparison2 = "java".compareTo("python");// This will be a -ve value -6
int comparison3 = "python".compareTo("java"); // This will be a +ve value 6
```

For example, ”java".compareTo(”python") returns a negative value because ‘j' comes before ‘p' in Unicode order. Similarly, " python".compareTo(”java") returns a positive value because ‘p' comes after ‘j'. If the strings are the same, it returns 0.

Fetch a character at an **index** in String

eazy bytes

To retrieve a character at a specific position within a String object, you can utilize the **charAt()** method. The indexing begins at zero. In the string "JAVA," the first character 'J' is at index 0, the second character 'A' is at index 1, and so on. It's important to note that the index of the last character 'A' is 3, which corresponds to the length of the string "JAVA" minus 1.

Character	J	A	V	A
Index position	0	1	2	3

```
String java = "JAVA";
char j = java.charAt(0); // J
```

If we try to fetch a character from the index position which is not available inside String value, then we will receive an **StringIndexOutOfBoundsException** due to which the program will be terminated.

Check if a String is **empty**

eazy bytes

To determine if a String is empty (contains no characters), you have three options:

1) Using isEmpty() method

This method specifically checks if the length of the string is zero.

```
String myString = "";
boolean isEmpty = myString.isEmpty(); // true
```

2) Using the length of the String

You can get the length of the string and check if it's zero.

```
String myString = "";
boolean isEmpty = myString.length()==0; // true
```

3) Using the equals() method

This method compares the content of the string to an empty string. It's preferred because it gracefully handles cases where the string is null.

```
String myString = "";
boolean isEmpty = "".equals(myString); // true
```

Among these methods, the third approach (""".equals(myString)) is often preferred because it's more robust. It doesn't throw an exception if the string is null and still correctly identifies an empty string. For example, """.equals(null) returns false, which is more predictable than potentially encountering a NullPointerException.

Changing the case in String

eazy bytes

To change the letters in a string to uppercase or lowercase in Java, you can use the **toUpperCase()** method for uppercase and **toLowerCase()** method for lowercase. For example, if you have the string "Java", applying toUpperCase() will give you "JAVA", and toLowerCase() will give you "java".

```
String originalString = "Java"; // contains "Java"
String uppercaseString = originalString.toUpperCase(); // contains "JAVA"
String lowercaseString = originalString.toLowerCase(); // contains "java"
```

It's important to remember that strings in Java are unchangeable (immutable). This means that when you use toUpperCase() or toLowerCase() on a string, the original string doesn't change. Instead, Java creates a new string with the modified case, leaving the original string unchanged.

We can write below code as well and it may looks like the String is mutable here. But behind the scenes, String will create the new content in a new memory location and the same will be referred by the variable,

```
String originalString = "Java"; // contains "Java"
originalString = originalString.toUpperCase(); // contains "JAVA"
```

Convert values as String

eazy bytes

In Java, the **String.valueOf()** method is a static method belonging to the String class, and it is used to convert different types of data, such as primitive types or objects, to their string representations. The primary purpose of this method is to create a string representation of the given data.

```
int intValue = 42;
double doubleValue = 3.14;
boolean boolValue = true;

String str1 = String.valueOf(intValue);
String str2 = String.valueOf(doubleValue);
String str3 = String.valueOf(boolValue);

System.out.println(str1);   // "42"
System.out.println(str2);   // "3.14"
System.out.println(str3);   // "true"
```

Using **concatenation operator (+)** as well, we can convert a given data into String. Below are examples,

```
int age = 30;
String message = "My age is: " + age;
String piValue = "" + 3.14;
```

Searching for a value in String

eazy bytes

In Java, searching for a value or a substring within a string can be accomplished using various methods provided by the String class. Here are a few common approaches:

1. indexOf() Method: The indexOf() method is used to find the index of the first occurrence of a specified substring within a string. If the substring is not found, it returns -1.

```
String originalString = "Hello, World!";
int index = originalString.indexOf("World");
System.out.println("Substring found at index: " + index);
```

2. contains() Method: The contains() method checks whether a specific substring is present in the given string. It returns a boolean value.

```
String originalString = "Hello, World!";
boolean isContains = originalString.contains("World"); // true
boolean isPresent = originalString.contains("@"); // false
```

3. startsWith() and endsWith() Methods: The startsWith() and endsWith() methods are used to check if a string starts or ends with a specified prefix or suffix.

```
String originalString = "Hello, World!";
boolean startsWithHello = originalString.startsWith("Hello"); // true
boolean endsWithWorld = originalString.endsWith("!"); // true
```

Searching for a value in String

eazy bytes

4. matches() Method: The matches() method can be used for more complex pattern matching using regular expressions.

```
String originalString = "Hello, World!";
boolean isMatched = originalString.matches(".*World.*"); // true
```

The methods mentioned above are case-sensitive. If you need a case-insensitive search, you can convert both the original string and the search pattern to lowercase (or uppercase) using toLowerCase() or toUpperCase().

Choose the method that best fits your specific use case and requirements. Each method has its advantages depending on the situation, and the choice often depends on the specific logic you need for your application.

Trimming a String

eazy bytes

The **trim()** method in Java is used to get rid of any extra spaces or control characters at the beginning and end of a string. Specifically, it removes characters with Unicode values less than \u0020 (decimal 32).

For instance:

" java ".trim() will result in the string "java" without the leading & trailing spaces.
”java ".trim() will return ”java" by removing the trailing space.
"\n \t java \n\r".trim() will become ”java" as it removes leading & trailing newline, carriage return, and tab characters.

```
String java1 = " java ".trim(); //java
String java2 = "java ".trim(); //java
String java3 = "\n \t java \n\r".trim(); //java
```

It's important to note that trim() only deals with spaces and control characters at the start and end of the string. It won't remove spaces or control characters if they are in the middle of the string. For example:

" ja\nva ".trim() will give "ja\nva" because the newline character (\n) is inside the string.
”j ava".trim() will result in "j ava" since the space is within the string and not at the beginning or end.

```
String java1 = " ja\nva ".trim() ; //ja\nva
String java2 = "j ava".trim(); //j ava
```

Fetching **Substring** from a String

eazy bytes

In Java, the substring method is used to extract a portion of a string. It allows you to create a new string that consists of a specific range of characters from the original string. The substring method has a couple of overloaded forms.

Basic Syntax:

startIndex: The index from which the substring begins (inclusive).
endIndex: The index where the substring ends (exclusive)

```
String originalString = "Hello, World!";
String substring1 = originalString.substring(startIndex);
String substring2 = originalString.substring(startIndex, endIndex);
```

In the below first example, the substring starts from index 7 ("W") to the end of the string.

Where as in second example, the substring starts from index 0 ("H") and ends at index 5(","), but the character at index 5 is not included in the result.

```
String originalString = "Hello, World!";
String substring1 = originalString.substring(7); // "World!"
String substring2 = originalString.substring(0, 5); // "Hello"
```

Important Points:

- Indexing in Java starts from 0. So, the first character of a string is at index 0.
- Negative indices or an endIndex less than the startIndex will result in an exception (StringIndexOutOfBoundsException).
- The substring method does not modify the original string. Instead, it creates and returns a new string.
- The length of the resulting substring is equal to endIndex - startIndex.

Replacing a part of a String

eazy bytes

In Java, the replace method is used to replace occurrences of a specific character or sequence of characters in a string with another character or sequence. This method comes in a few different forms to cater to different replacement scenarios.

Basic Syntax:

oldChar: The character or sequence of characters to be replaced.
newChar: The character or sequence of characters to replace the old ones.

```
String originalString = "Hello, World!";
String replacedString = originalString.replace(oldChar, newChar);
```

In this example, all occurrences of the character 'o' in the string are replaced with '*'.

```
String originalString = "Hello, World!";
String replacedString = originalString.replace('o', '*'); // "Hell*, W*rld!"
```

Replace a Substring

In this example, the substring "World" is replaced with "Universe".

```
String originalString = "Hello, World!";
String replacedString = originalString.replace("World", "Universe");
// "Hello, Universe!"
```

Note that the replace method is case-sensitive.

Replacing a part of a String

eazy bytes

Replace All Occurrences:

```
String originalString = "abababab";
String replacedString = originalString.replace("ab", "X"); // "XXXX"
```

The **replaceAll(String regex, String replacement)** method uses a regular expression in regex to find matches. It returns a new String object by replacing each match with replacement. This example replaces all vowels with '*'.

```
String originalString = "Java is fun!";
String replacedString = originalString.replaceAll("a|e|i|o|u", "*");
// "J*v* *s f*n!"
```

The **replaceFirst(String regex, String replacement)** method works the same as the replaceAll() method, except that it replaces only the first match with given replacement.

```
String originalString = "apple orange apple banana apple";
// Replace the first occurrence of "apple" with "grape"
String replacedString = originalString.replaceFirst("apple", "grape");
// grape orange apple banana apple
```

Splitting Strings

eazy bytes

In Java, the split method is used to split a string into an array of substrings based on a specified delimiter. Here's a basic explanation.

Basic Syntax:

```
String[] parts = originalString.split(delimiter);
```

originalString: The string you want to split.

delimiter: The character or regular expression used to determine where to split the string.

In this example, the split method is used to split the string "apple,orange,banana,grape" into an array of strings using ',' as the delimiter. After splitting, fruitArray will contain ["apple", "orange", "banana", "grape"].

```
String fruits = "apple,orange,banana,grape";
String[] fruitArray = fruits.split(",");
```

Joining Strings

eazy bytes

The String.join method was introduced in Java 8. It is designed to concatenate a list of strings with a specified delimiter. The basic syntax is as follows:

Basic Syntax:

```
String result = String.join(delimiter, elements);
```

delimiter: The character or sequence of characters that will be used to separate the elements in the resulting string.
elements: The strings to be joined.

In this example, the strings "Hello", "World", and "Java" are joined into a single string with a space (' ') as the delimiter. The output will be like shown below:

```
String result = String.join(" ", "Hello", "World", "Java");
//Hello World Java
```

Joining Strings

eazy bytes

If you want to concatenate strings without any delimiter, you can use an empty string ("") as the delimiter. The output will be shown as below,

```
String result = String.join("", "Java is", "fun and", "powerful");
//Java isfun andpowerful
```

Even if the list is empty, String.join gracefully handles it, and the output will be an empty string.

```
List<String> emptyList = Collections.emptyList();
String result = String.join(", ", emptyList);
```

If any element in the list is null, it will be treated as the string "null" in the resulting string.

String.join internally uses a StringBuilder for concatenation, making it efficient for joining multiple strings.

format() method in String

eazy bytes

The **String.format** method in Java is a powerful tool for creating formatted strings. It allows you to construct strings with placeholders for various types of values and control the formatting of those values. The basic syntax of String.format is as follows:

```
String formattedString = String.format(format, arg1, arg2, ...);
```

format: A format string that contains placeholders for the arguments.
arg1, arg2, ...: Arguments to be formatted and inserted into the placeholders.

The format string specifies how the arguments should be formatted. It can contain placeholders for the arguments, which are denoted by % followed by a format specifier. The format specifier specifies how the argument should be formatted. In the below example, the format string contains two placeholders: %s for the name, and %d for the number of messages. The **String.format** method takes the format string and the values of the name and numMessages variables, and returns a new string with the placeholders replaced by the formatted values. The resulting string would be:

```
String message = "Hello, %s ! You have %d new messages.";
String name = "Madan";
int numOfMessages = 3;
String formattedMessage = String.format(message, name, numOfMessages);
// Hello, Madan ! You have 3 new messages.
```

format() method in String

eazy bytes

The format string can contain placeholders, each denoted by a % character followed by a conversion specifier. Here are some common conversion specifiers:

%s: String
%d: Integer
%f: Floating-point number
%c: Character
%b: Boolean

When using format() method, you can control the precision of the floating numbers like shown below,

```
double price = 19.99;
// The price is $19.990000
String formattedPrice = String.format("The price is $%f", price);
// The price is $19.99
String formattedPrice = String.format("The price is $%.2f", price);
```

format() method in String

eazy bytes

When using format() method, you can apply the padding/alignment like shown below. Here, %5d specifies that the integer should be padded with spaces to a width of 5.

```
int number = 42;

String paddedNumber = String.format("The number is %5d", number);
// The number is    42
```

You can use the argument index to reference arguments in a different order. For example, in the below example, using %2$s and %1$d, we are trying to refer the arguments in the order that we like,

```
String name = "John";
int age = 25;
String formattedString = String.format("My name is %2$s, I am %1$d years
old, and I live in %3$s.", age, name, "City");
// My name is John, I am 25 years old, and I live in City.
```

System.out.printf() method

eazy bytes

In Java, the println method is used to print a line of text to the console. When you want to format a string and then print it using println, you can use the printf method to achieve the desired formatting. Below are few examples of printf(),

```
System.out.printf("Name: %s, Age: %d%n", "John", 25);
```

```
// Format the floating-point number with two decimal places
System.out.printf("Price: %.2f%n", 19.99);
```

```
System.out.printf("Grade: %c%n", 'A');
```

```
System.out.printf("Is Java fun? %b%n", true);
```

String is **immutable**

eazy bytes

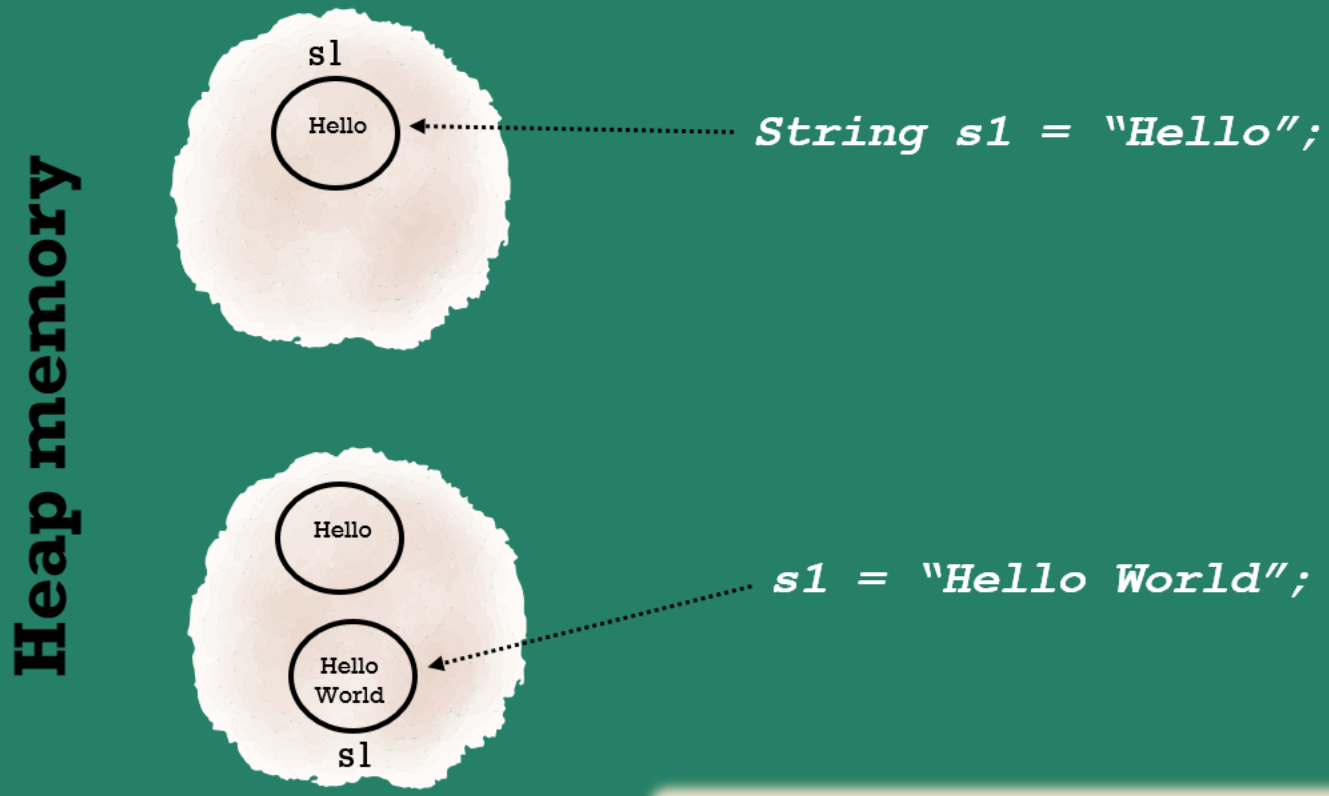
> The String class is **immutable**. Just like we can't change tatoos once created, similarly once a String object is created in Java, its contents cannot be altered.
>
> However, there are two mutable companion classes for the String class: **StringBuilder** and **StringBuffer**. These classes should be employed when there is a need to modify the contents of a string.

You may have illusion, that String is allowing to change the value like shown below. But behind the scenes, a new memory location will be pointed for the new String value.

```
String s1 = "Hello";
s1 = "Hello World";
```

String is **immutable**

eazy bytes



> The immutability applies to the String object stored in memory, not the reference variable of the String type. If you intend to have a reference variable consistently point to the same String object in memory, you need to declare the reference variable as final

```
final String s1 = "Hello";
s1 = "Hello World"; // Compilation error as s1 is marked as final
```

How to create **mutable strings** in Java

eazy bytes

In Java, the String class is immutable, which means that once a String object is created, its value cannot be changed. However, there are several ways to create mutable strings in Java:

1. **StringBuilder** : This class is designed for building and manipulating strings. It provides various methods for appending, inserting, replacing, and deleting characters in a string. Here's an example:

```
StringBuilder sb = new StringBuilder("hello");
sb.append(" world");
System.out.println(sb.toString());  // "hello world"
```

2. **StringBuffer** : This class is similar to StringBuilder and provides the same functionality for manipulating strings. The only difference is that StringBuffer is **thread-safe**, whereas StringBuilder is not. Here's an example:

```
StringBuffer sb = new StringBuffer("hello");
sb.append(" world");
System.out.println(sb.toString());  // "hello world"
```

Use StringBuilder when:
You are working in a single-threaded environment.
You need higher performance, and thread safety is not a concern.

Use StringBuffer when:
You are working in a multi-threaded environment where thread safety is critical.

How to create **mutable strings** in Java

eazy bytes

Both StringBuilder & StringBuffer supports various mutable operations using methods like append, insert, delete, replace

The StringBuilder & StringBuffer classes has a reverse() method, which replaces its contents with the same sequence of characters, but in reverse order.

```java
// Create an empty StringBuilder
StringBuilder sb = new StringBuilder();
sb.append("World"); // World
sb.insert(0, "Hello "); // Hello World
sb.deleteCharAt(4); // Hell World
sb.insert(4,'o'); // Hello World
sb.setLength(5); // Hello
sb.reverse(); // olleH
```

```java
// Create an empty StringBuffer
StringBuffer sb = new StringBuffer();
sb.append("World"); // World
sb.insert(0, "Hello "); // Hello World
sb.deleteCharAt(4); // Hell World
sb.insert(4,'o'); // Hello World
sb.setLength(5); // Hello
sb.reverse(); // olleH
```

How to create **mutable strings** in Java

eazy bytes

3 **CharArrayWriter :** This class allows you to write characters to a buffer and then convert them to a string. Here's an example:

```
CharArrayWriter cw = new CharArrayWriter();
cw.write("hello");
cw.write(" world");
String s = cw.toString();
System.out.println(s);  // "hello world"
```

Note that these mutable string classes provide similar functionality to the immutable String class, but with the ability to modify the contents of the string. However, it's important to use them only when necessary, as they can be less efficient than String for certain operations.

Text Block in Java

eazy bytes

Java 15 introduced a new type of string literal called a **text block**, which allows programmers to preserve indents and multiple lines without the need to add white spaces within quotes. Prior to this, programmers had to add indentation and use escape sequences like \n to break lines. Here is an example of how it was done before Java 15:

```
String htmlCode = "<html>\n" +
                  "    <body>\n" +
                  "        <p>Hello World.</p>\n" +
                  "    </body>\n" +
                  "</html>\n";
```

In Java, you can create a text block by enclosing the text in **triple quotes (""")** like shown below,

```
String htmlCode = """
                    <html>
                        <body>
                            <p>Hello World.</p>
                        </body>
                    </html>
                  """;
```

Text blocks make it much easier to define multiline HTML code, SQL queries or JSON bodies within Java programs.

From Java 15

Text Block in Java

eazy bytes

You can also use text blocks to create formatted strings, using placeholders like **%s** or **%d**. For example:

```
String name = "Alice";
int age = 30;
String message = """
    Hello, my name is %s
    and I am %d years old.
    """;
String formattedMessage = String.format(message, name, age);
System.out.println(formattedMessage);
```

This would output:

```
Hello, my name is Alice
and I am 30 years old.
```

Convert values as String

eazy bytes

In Java, the **String.valueOf()** method is a static method belonging to the String class, and it is used to convert different types of data, such as primitive types or objects, to their string representations. The primary purpose of this method is to create a string representation of the given data.

```
int intValue = 42;
double doubleValue = 3.14;
boolean boolValue = true;

String str1 = String.valueOf(intValue);
String str2 = String.valueOf(doubleValue);
String str3 = String.valueOf(boolValue);

System.out.println(str1);  // "42"
System.out.println(str2);  // "3.14"
System.out.println(str3);  // "true"
```

Using **concatenation operator (+)** as well, we can convert a given data into String. Below are examples,

```
int age = 30;
String message = "My age is: " + age;
String piValue = "" + 3.14;
```

String to Primitive data type values

eazy bytes

Parsing involves converting a string representation of a value into its corresponding primitive data type. In Java, this is commonly done using methods provided by wrapper classes. Let's go through parsing for various primitive data types:

Parsing Integer Values (int):

```
String strNumber = "123";
int parsedNumber = Integer.parseInt(strNumber); // 123
```

Parsing Double Values (double):

```
String strDouble = "3.14";
double parsedDouble = Double.parseDouble(strDouble); // 3.14
```

Parsing Floating-Point Values (float):

```
String strFloat = "2.718";
float parsedFloat = Float.parseFloat(strFloat); // 2.718
```

String to Primitive data type values

eazy bytes

Parsing Long Values (long):

```
String strLong = "3476543210";
long parsedLong = Long.parseLong(strLong); // 3476543210
```

Parsing Boolean Values (boolean): If you try mentioning an invalid value other than true/false, the result will be false.

```
String strBoolean = "true";
boolean parsedBoolean = Boolean.parseBoolean(strBoolean); // true
```

During parsing, if a valid number is not provided, the NumberFormatException will be thrown at runtime. So it is recommended to handle this exception inside the code. More to come around this, in coming sections.

String to Primitive data type values

eazy bytes

In addition to parsing into primitive data types directly, you can also use the constructor of the wrapper classes like shown below. But this approach is not recommended & deprecated from Java 9.

```
String strNumber = "456";
Integer integerObject = new Integer(strNumber); // 456
```

Converting a String to a char involves extracting a single character from the string. Since a String is a sequence of characters, you need to specify which character you want to extract. Here's how you can convert a String to a char using charAt():

```
String str = "Hello";
char firstChar = str.charAt(0); // H
```

The toCharArray() method converts the entire string into an array of characters. If you know the index of the character you want, you can access it from the array.

```
String str = "Hello";
char[] charArray = str.toCharArray();
char secondChar = charArray[1]; // e
```