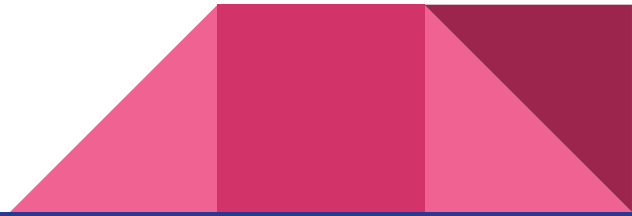
- Java Memory Architecture
- Strings

Java memory architecture

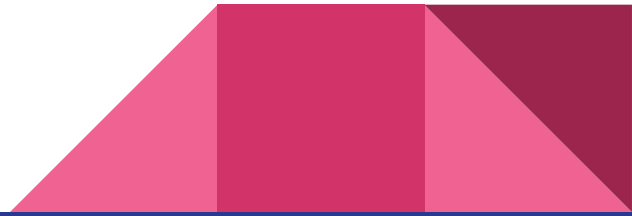
In Java memory architecture, the JVM (Java Virtual Machine) divides memory into different regions to manage how programs run.

Two of the most important areas are the **Stack** and the **Heap**.



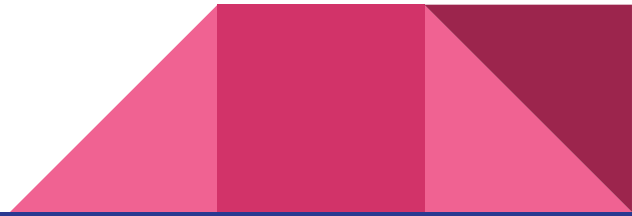
Stack

- Stores local variables (like primitives: int, double, char, etc.)
- Stores references (memory addresses) to objects in the heap
- Stores method calls (each method has its own "stack frame")
- Memory is allocated and freed automatically as methods are called and return
- Follows LIFO (Last In, First Out) order.

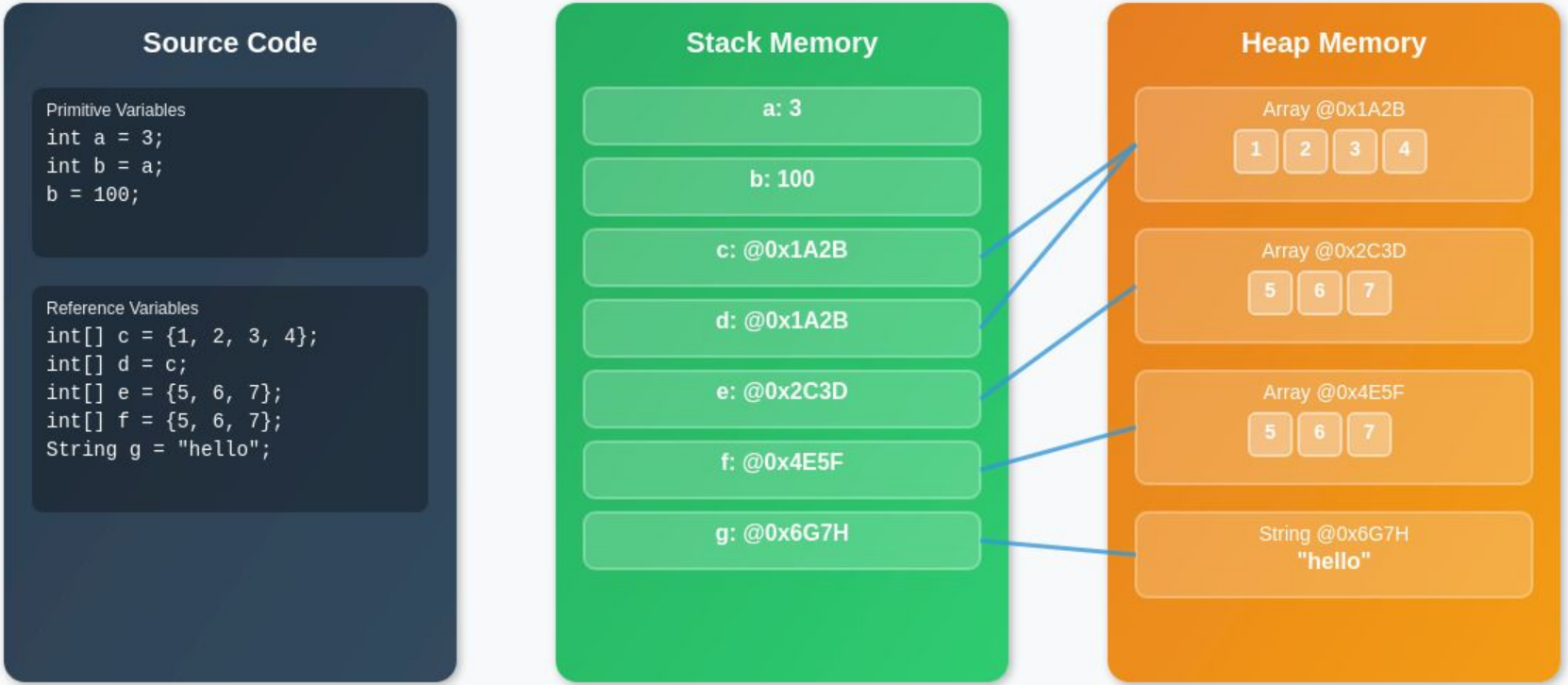


Heap

- Stores all objects (new keyword instances, arrays, Strings, etc.)
- Stores instance variables inside objects
- It has a special area called **String Pool** where Java stores string literals



Java Memory Architecture



Strings

A String in Java is a **sequence of characters.**

Strings are **Objects**, so they have **Methods**.

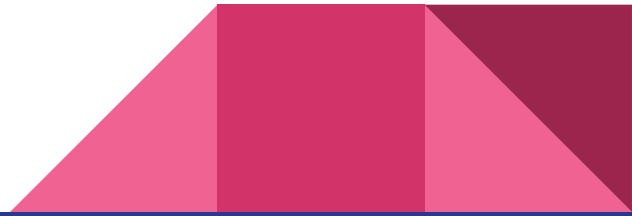
How should we create Strings?

1. We can use a **String literal**. Use **double quotes** to create a String.

   ```
   String name = "Sophie";
   ```

2. We can use the **new** operator.

   ```
   String name = new String("Sophie");
   ```



String Literal

String str1 = "abc";

- Java creates a String object with value "abc" in the String Pool
- str1 gets a reference pointing to this object

What happens if I create another string: String str2 = "abc";?

- Java checks: "Do I already have "abc" in the String Pool?"
- Answer: YES! It's already there from str1
- **str2** gets a **reference** to the **same existing object**
- **No new object is created**, this saves memory!

Using the **new** operator

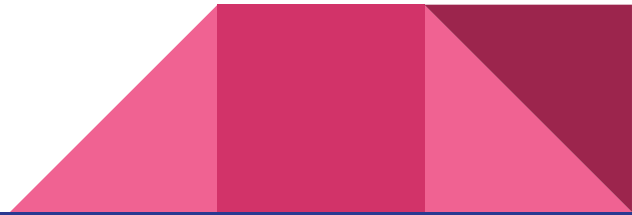
String str3 = new String("abc");

- The new keyword **new forces** Java to create a brand new String object

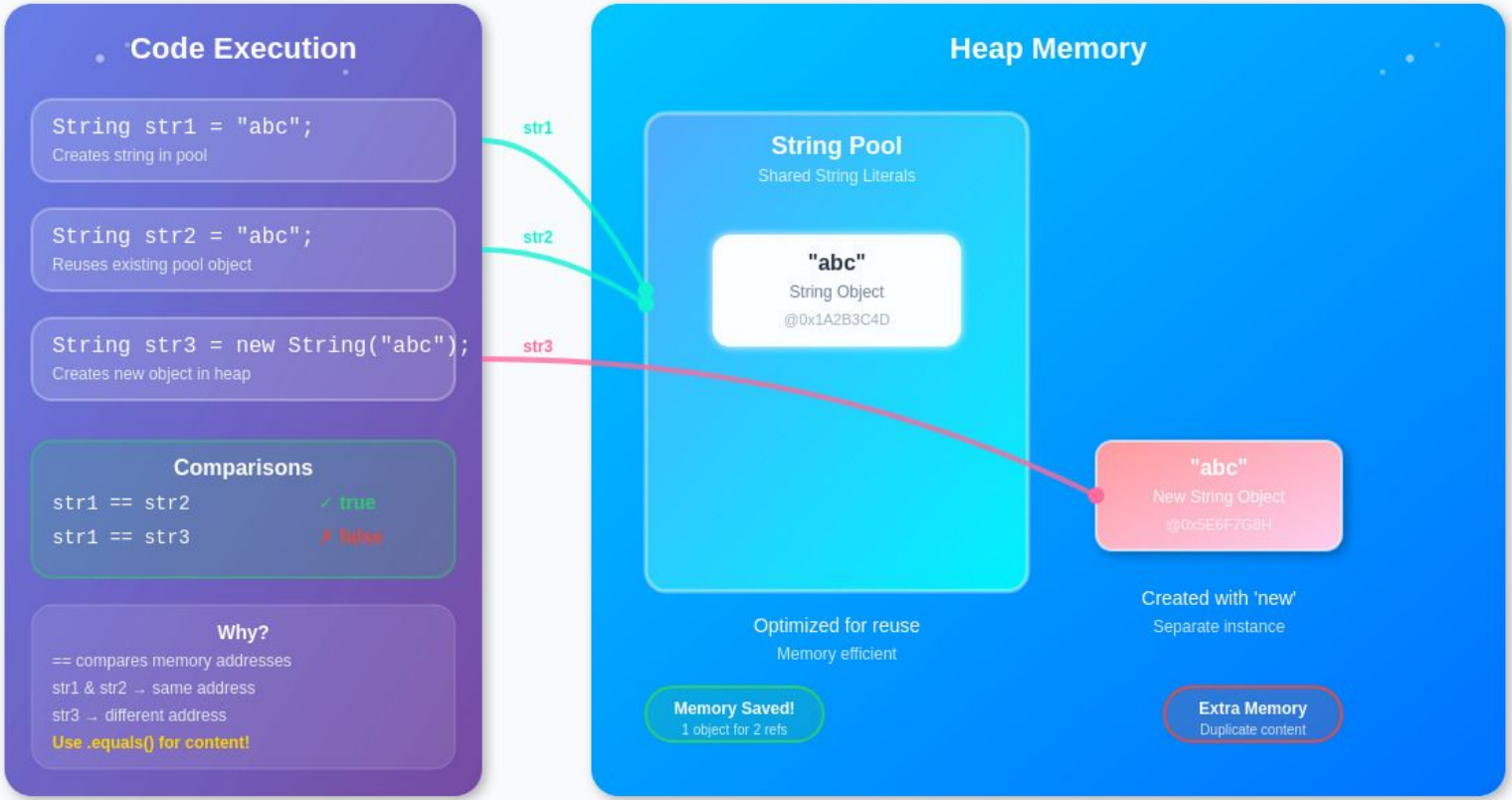
- This new object is created in the regular heap area (outside the String Pool)

- Even though the content is "abc" (same as str1 and str2) , it's a **completely separate object**

- str3 points to a new object



Java String Pool Memory



String methods

Method	Use
name.**substring(int start, int end)**	returns a copy of the string between the two indices excluding the end
name.**substring(int start)**	returns a copy of the string starting at the index, up until the end
name.**equals(Object another)**	return true if the strings have identical contents
name.**length()**	returns the number of characters in str
name.**compareTo(String another)**	for less than/ greater than / equal comparison
name.**charAt(int index)**	return the character at the index position of the String

More about class String:
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html

Immutability

String methods do **not** alter the existing String, they create **new** ones.

String name = “sophie”;

String newName = name.toUpperCase();

System.out.println(name); - - - - - - - - - -> sophie

System.out.println(newName); - - - - - - - - - -> SOPHIE

Immutability

Strings are **immutable** which means they cannot change once they are created.

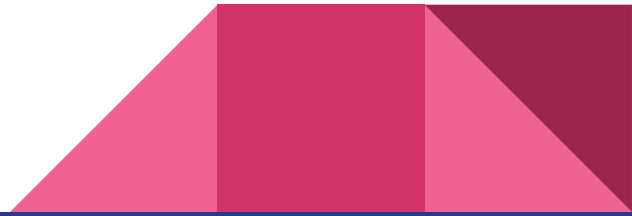
The only way to **change** the value of name is to **re-assign** it:

String name = “sophie”;

System.out.println(name); - - - - - - - - - -> sophie

name = “SOPHIE”;

System.out.println(name); - - - - - - - - - -> SOPHIE



Concatenation

Strings can be **concatenated** with one another to create a **new String** value.

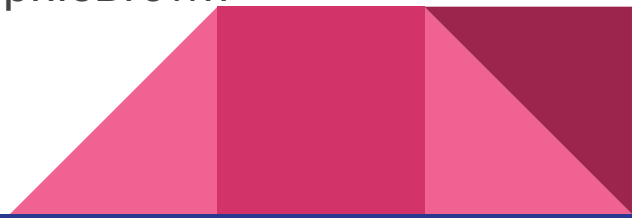
Concatenate: Add 2 string values together.

String firstName = “Sophie”;

String lastName = “Brown”;

String fullName = firstName + lastName;

System.out.println(fullName); - - - - - - - - -> SophieBrown



Concatenation

String firstName = “Sophie ”;

String lastName = “Brown”;

String fullName = firstName + lastName;

System.out.println(fullName); - - - - - - - - -> Sophie Brown

Concatenation - Shortcut

The shortcut **+=** works on String values:

String name = “Sophie ”;

name += “Brown”;

System.out.println(name); - - - - - - - - -> Sophie Brown

Concatenating Primitives

Primitive Types can be concatenated with String objects:

String name = “Sophie”;

int age = 8;

System.out.println(name + “ is “ + age); - - - - - - - - ->   Sophie is 8

The primitive type is converted to String, this is called **implicit conversion.**

Concatenating Primitives

Implicit conversion can be tricky.

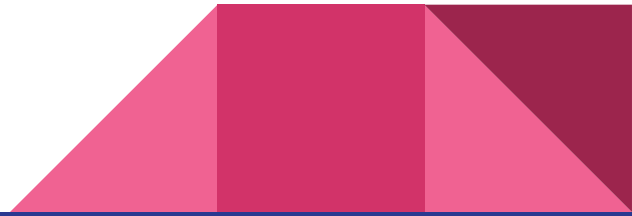
What do you think the outcome of this program is?

```
int currentAge = 20;

int age = 10;

System.out.println(“In ten years, I will be: “ + currentAge + age);
```

In ten years, I will be: 2010



Concatenating Primitives

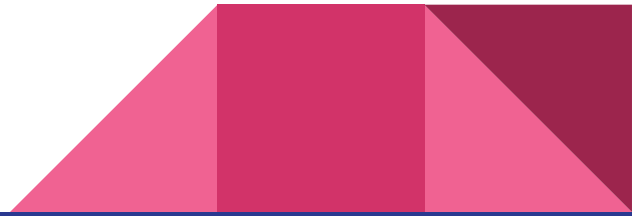
How can we make it work without using an extra variable?

```
int currentAge = 20;

int age = 10;


System.out.println("In ten years, I will be: " + (currentAge + age));
```

In ten years, I will be: 30



String dilemma

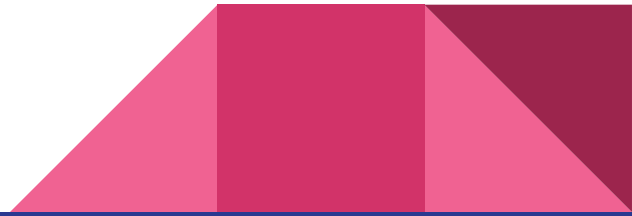
Some characters cannot be used directly because they hold a specific meaning in Java.

Example:

Include quotes in a line of code, will cause an error:

String str = ”She said, “Hello!”” - - - - - -> “Hello!” is interpreted as a String value

- - - - - -> The compiler thinks Hello! Is a variable name



Escape Sequences

We can include “” in our code by writing a **escape sequence \”**

Example:

String str = “She said, **\”**Hello!**\”**”;

System.out.println(str); - - - - - -> She said, “Hello!”

Some useful escape sequences

Escape Sequences allow us to include special characters and actions in String objects.

Escape Sequence	Function	Output
\”	“ \” Allow for quotations\” “	“Allow for quotations”
\\	“Includes a backslash\\”	Includes a backslash\
\n	This creates \na line break	This creates a line break
\t	“This adds a \ttab space”	This adds a tab space

Let’s practice

Classwork on GitHub