

Intro to Objects with Strings

Module 1: 06

Week 2 Overview

Monday

Intro to
Objects with
Strings

Tuesday

Collections
Part 1

Wednesday

Collections
Part 2

Thursday

Classes and
Encapsulation

Friday

Review

Today's Objectives

- Objects and Classes
- Creating Objects
- Data Types in Memory
 - Value and Reference Types
 - Stack and Heap
- Strings
- Immutability
- null
- Reference and Value Equality
- String Methods

Object Terms and Definitions

State

The *current* values of data being stored in an object. Usually in variables.

Behavior

What an object can do. Provided by methods.

Memory

A physical device (RAM) used to store information and programs for immediate use.

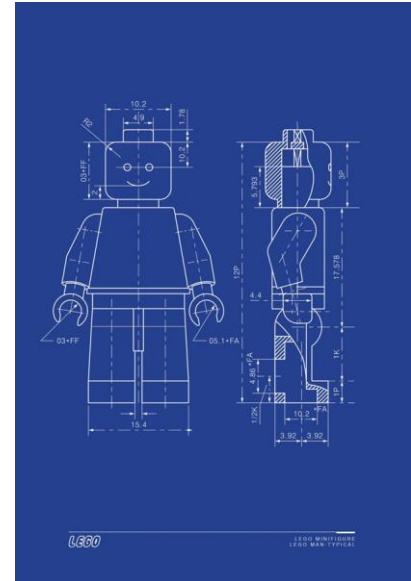
Abstraction

The process of removing characteristics from something in order to reduce it to a set of *essential characteristics* (state and behavior).



Class

- **Source code** that defines how to create an object and what state and behavior that object will have.
- **A class is a blueprint for an object.**
- **Classes are source code** that give instructions, but don't exist when the programming is running.
- **Classes in Java Define a Data Type**, which makes everything in Java a Data Type.



Objects

An **object** is an *in-memory data structure* that combines **state** and **behavior** into a usable and useful abstraction.

- **Objects are not in source code**
- **Objects only exist in memory** while a program is running
- Multiple Objects can be created from a Class, but **each object is distinct and separate** from every other object in our program



Class

Objects

recipe: favorite chocolate cake

ingredients

1 pkg. devil's food cake mix	1/2 C. warm water
1 pkg. instant chocolate pudding	1/2 C. oil
4 large eggs	1 1/2 C. semi-sweet chocolate chips
1 C. sour cream	

instructions

Grease a fluted tube pan and preheat oven to 350 degrees. Mix ingredients together except chocolate chips, using a mixer. Fold in chocolate chips.

Bake for 45 - 50 minutes, or until inserted toothpick comes out clean. Cool for 20 minutes and then invert and remove from pan.

Dust with powdered sugar and serve with fresh whipped cream and strawberries or raspberries.

bake



bake



bake



Class

```
public class Cake {  
  
    private String icingColor;  
    private boolean hasSprinkles;  
  
    public String getIcingColor() {  
        return icingColor;  
    }  
  
    public void setIcingColor(String icingColor)  
    {  
        this.icingColor = icingColor;  
    }  
  
    public boolean isHasSprinkles() {  
        return hasSprinkles;  
    }  
  
    public void setHasSprinkles(boolean  
hasSprinkles) {  
        this.hasSprinkles = hasSprinkles;  
    }  
}
```

new

```
≡ vanillaCake = {Cake@798}  
> f icingColor = "Vanilla"  
f hasSprinkles = true
```



new

```
≡ chocolateCake = {Cake@798}  
> f icingColor = "Chocolate"  
f hasSprinkles = true
```



new

```
≡ strawberryCake = {Cake@798}  
> f icingColor = "Strawberry"  
f hasSprinkles = false
```



Class vs Object

Class	Object
A template for creating (<i>instantiating</i>) objects within a program	An <i>instance</i> of a class
Logical entity	Physical entity
Exists only in source code	Exists only in memory when the program is running
Declared with class keyword	Created using the new keyword
Declared once	Multiple distinct objects can be created using a class

Everything in Java, except the 8 primitive data types (char, byte, short, int, long, float, double, boolean), **is an object and has a class that defines it.**

Everything in Java is a Data Type!

Creating Objects

An *Object* is an **instance** of a *class*.

Steps to **instantiate** (create) an *object* from a *class*:

1. **Declare** a variable to hold the object. The class will be the data type.
2. **Instantiate** a new object from the class using the **new** keyword
3. **Initialize** variables inside the object with initial values using the *constructor*

```
Cake chocolateCake = new Cake("Chocolate", true);
```

```
Scanner in = new Scanner(System.in);
```

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

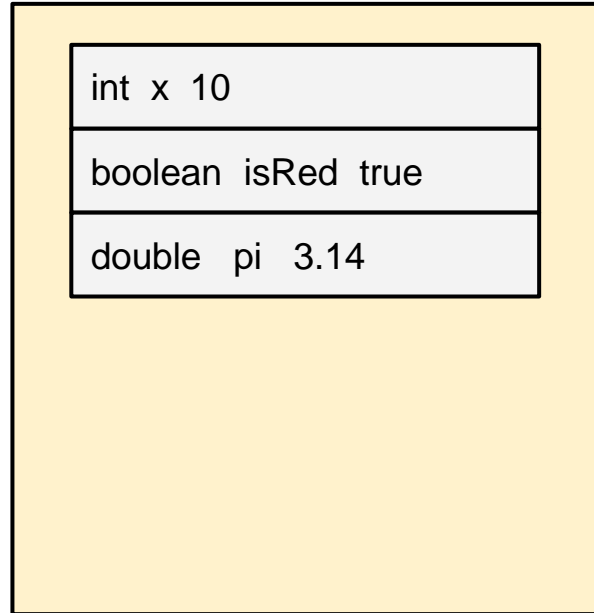
Data Types Memory

2 types of Data Type in Java

1. **Value Type** (*primitive*) reference a single static space in memory to hold the value. This memory is allocated on the **stack**.
 - a. Only 8 value types: byte, char, short, int, long, float, double, and boolean
2. **Reference Type** (*object*) does not hold the value in static memory, the *stack*, but holds a *reference* to where the object is located in free-floating, dynamic memory, the **heap**
 - a. Everything except the 8 value types is a reference type.
 - i. Examples: Scanner, String, Array

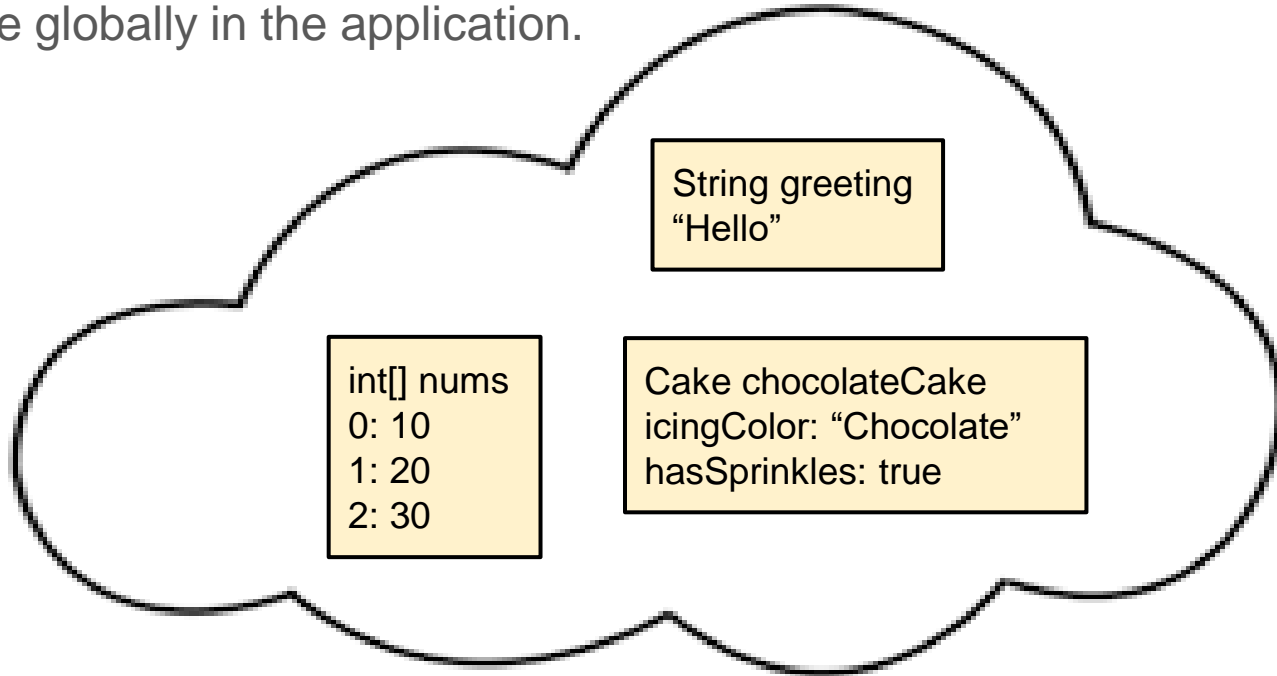
Stack memory

A linear structured area of computer memory *of a set size* that temporarily stores local application values. When a value is no longer needed it is automatically erased. The stack is managed by the OS. *The access to the stack is very fast.*



Heap memory

A free floating memory area that is allocated and deallocated (cleaned up) by programmers or the language (Java). The Heap can hold values of any size and are available globally in the application.



Stack vs Heap

Stack	Heap
A linear data structure	A free floating range of memory
High Speed Access	Slower access
Stores simple values of a set size (Primitive Value Types)	Stores complex objects of any size (Reference Types)
Managed by the OS	Managed by the programmer and language
Limited in Size	Unlimited in Size
Fixed Size	Can be resized
Main problem is running out of memory (StackOverflow)	Main problem is memory fragmentation (slower access)

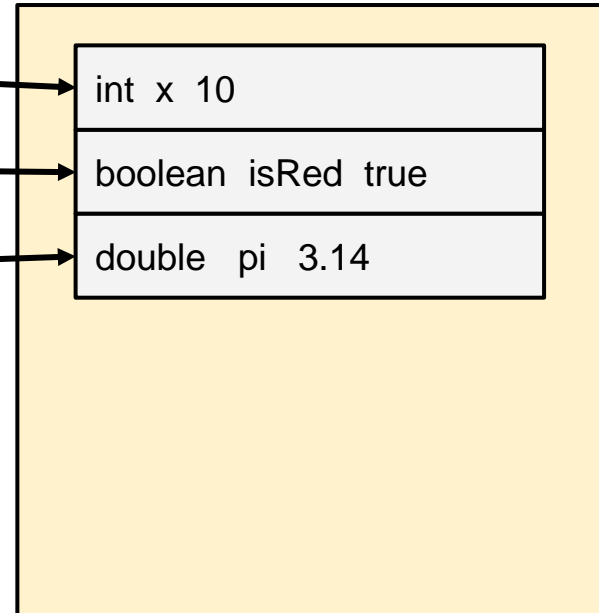
Value Types (Primitives) in Memory

Primitive data types (char, byte, short, int, long, float, double, boolean) are stored on the Stack

int x = 10;

boolean isRed = true;

double pi = 3.14;



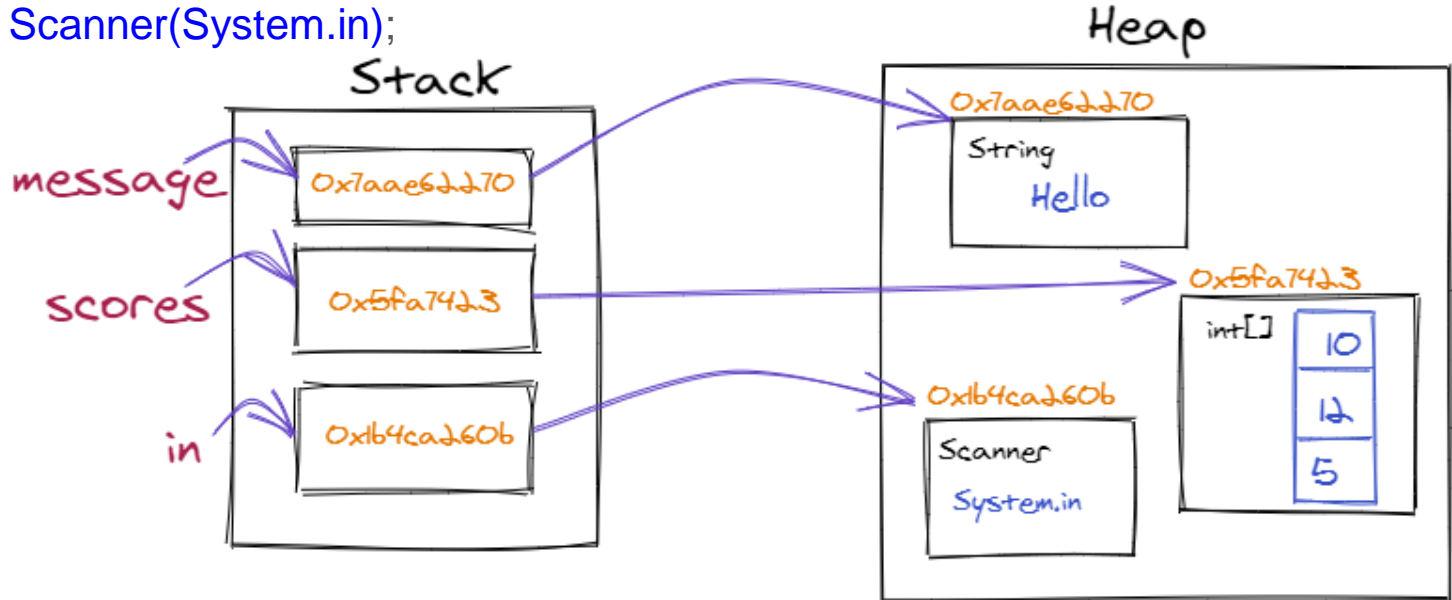
Reference Types (objects) in Memory

Reference Types (objects) are stored on the Heap. The variable stores a pointer (memory address) on the stack that points to the location of the object in the Heap. Access to the Heap is slower than the stack.

```
String message = "Hello";
```

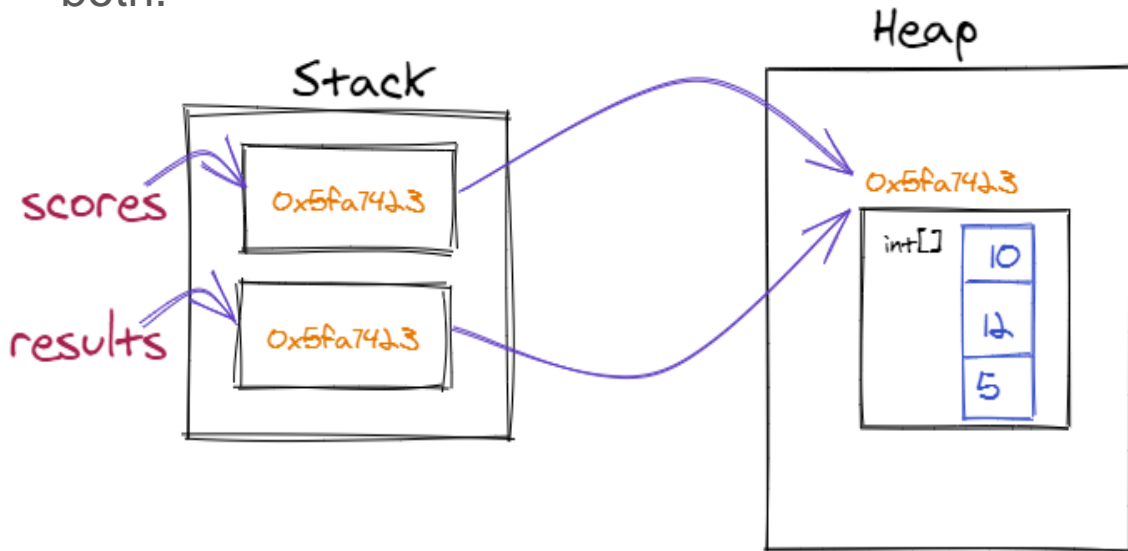
```
int[] scores = new int[3];
```

```
Scanner in = new Scanner(System.in);
```



Multiple variables can have the same Pointer

Since variables for Reference Types point to a value on the stack that contains a memory address that points to an object on the heap, it is possible for multiple variables to hold the same memory address, and thus point to the same object on the heap. This results in a change using either variable affecting the value for both.



```
int[] scores = new int[] {10, 12, 5};
```

```
int[] results = scores;
```

```
results[0] = 20;
```

```
results[0] == 20;
```

```
scores[0] == 20;
```

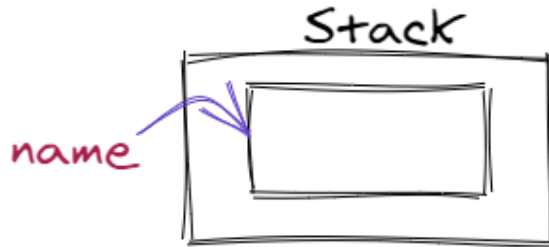
null

When variables for Reference Types are created they default to ***null***, until they are assigned a reference to an instantiated object.

null is not the same thing as ***empty***. ***null*** refers to no value on the Stack. ***empty*** refers to the value of an existing object on the Heap.

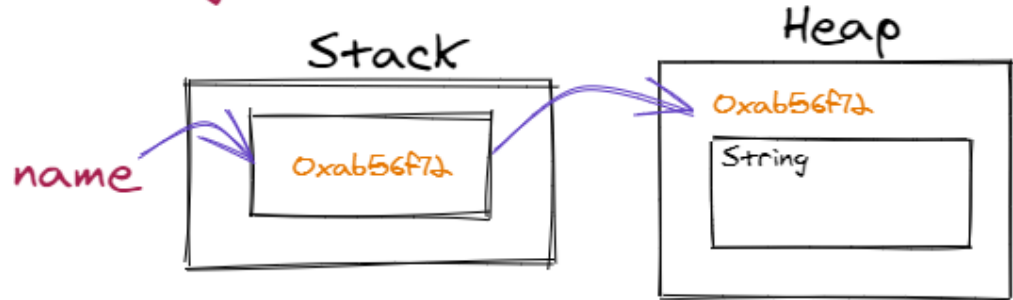
null

String name;



empty

String name = "";



No Value vs Null

0 / empty

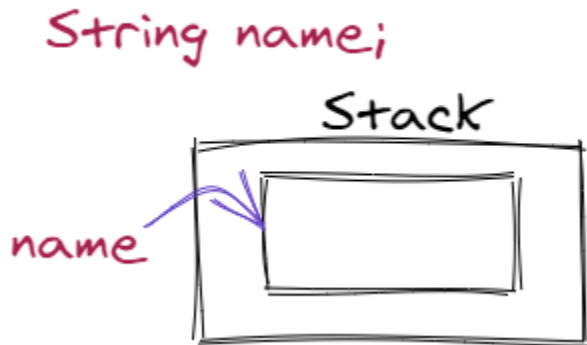


null



nullPointerException

A `nullPointerException` occurs when trying to use a method on an object when the variable does not contain a reference and is null.



```
String name;
```

```
name.toUpperCase(); ← nullPointerException
```

```
int[] numbers;
```

```
numbers.length; ← nullPointerException
```



There is no other cause of this exception.

Is Java pass by Reference or Value?

All programming languages are partially defined by whether they pass variables to methods by Value, meaning they pass the value of the variable, or by Reference, meaning they pass the memory location of the value.

Java is *Pass by Value*.

Java passes the value on the Stack when passing a variable to a method.

Common Interview Question

Creating Objects Revisited

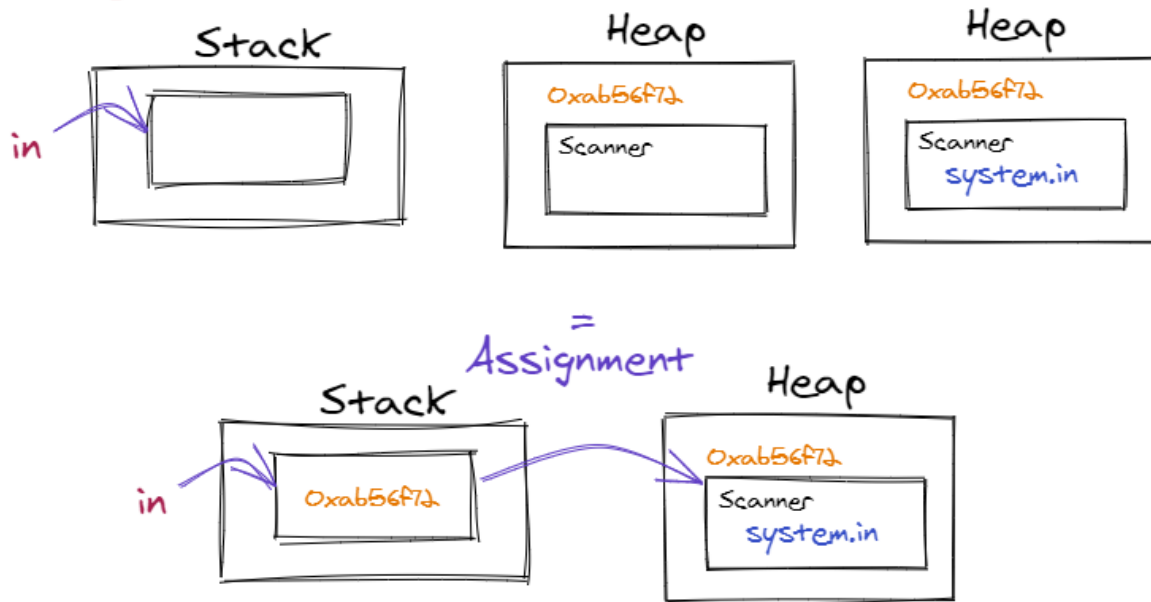
Declare a variable, *instantiate* a new Object using the *new* keyword, *initialize* the object with its starting state, and *assign* it to the variable

```
Scanner in = new Scanner ( System.in );
```

Scanner in
Declaration

new Scanner
Instantiation

(System.in)
Initialization



Strings

String is a References Type (Object), but for performance reasons String receives special treatment in Java that allows to to act in some ways like a primitive in code, but not in memory.

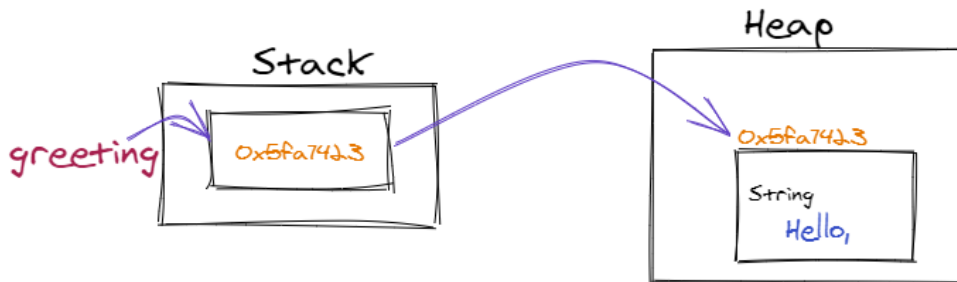
Can be instantiated with a literal	String greeting = "Hello";
Can use the + operator for concatenation	greeting = greeting + " John";
Is Immutable	Once created, the value of a String cannot be changed.
Uses a Common String Pool	If multiple Strings are created with the same literal (e.g. "Hello"). Only one common value is stored.
Stored as a char[]	The characters in a String are stored internally as an array of type char

Immutability

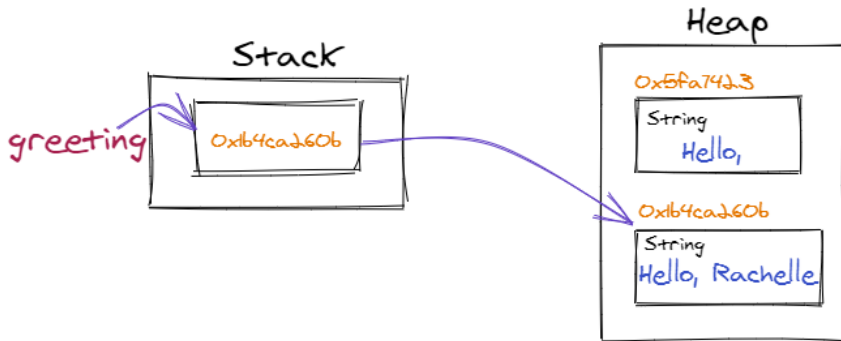
An object whose state cannot be changed after it is created.

String is immutable, meaning after a string is created the value cannot be changed, but instead a new String must be created with the new value.

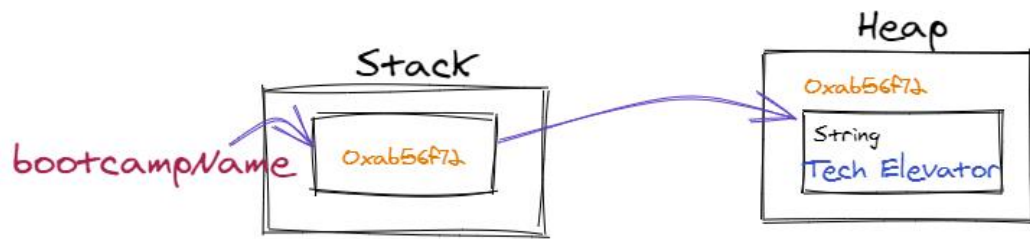
String greeting = "Hello , ";



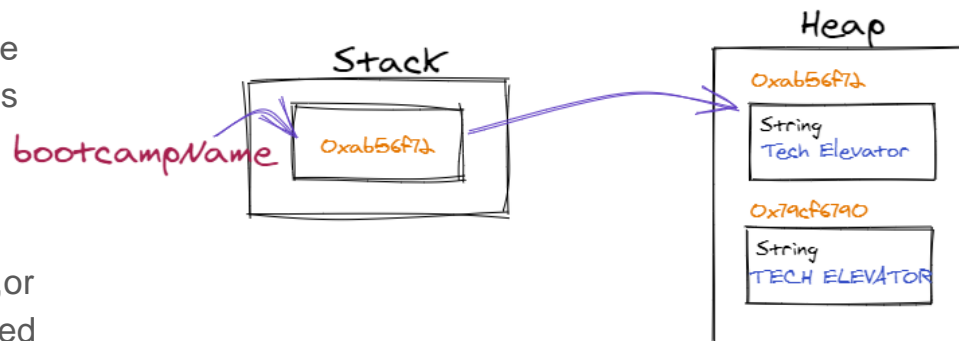
greeting = greeting + "Rachelle";



```
String bootcampName = "Tech Elevator";
```

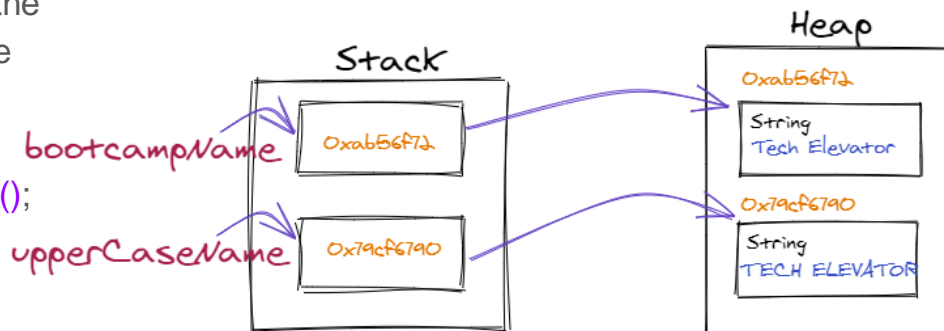


`bootcampName.toUpperCase();` ← does not change the original String, instead creates a new one, but the value is in an unreferenced object, so is lost.



When the value of a String, is changed by concatenation, or a String method, the resulting new String must be assigned to a variable, either the original variable or a new one so the location of the new object that contains the changed value is not lost.

```
String upperCaseName = bootcampName.toUpperCase();
```



Strings are a char[]

Since String holds the characters internally as a char[], it can be instantiated and initialized with a char[] to create a new String.

```
char[] awesomeArray = new char[] { 'A', 'w', 'e', 's', 'o', 'm', 'e'
};
String awesomeString1 = new String( awesomeArray );
```

awesomeString1 has a value of “Awesome”

Object Equality

```
String hello = new String( "Hello" );  
String hello2 = new String( "Hello" );
```

`hello == hello2` is **FALSE**

The value of Reference Types (objects) cannot use `==` to compare the value of objects, instead they use the `.equals()` method;

`hello.equals(hello2)` is **TRUE**

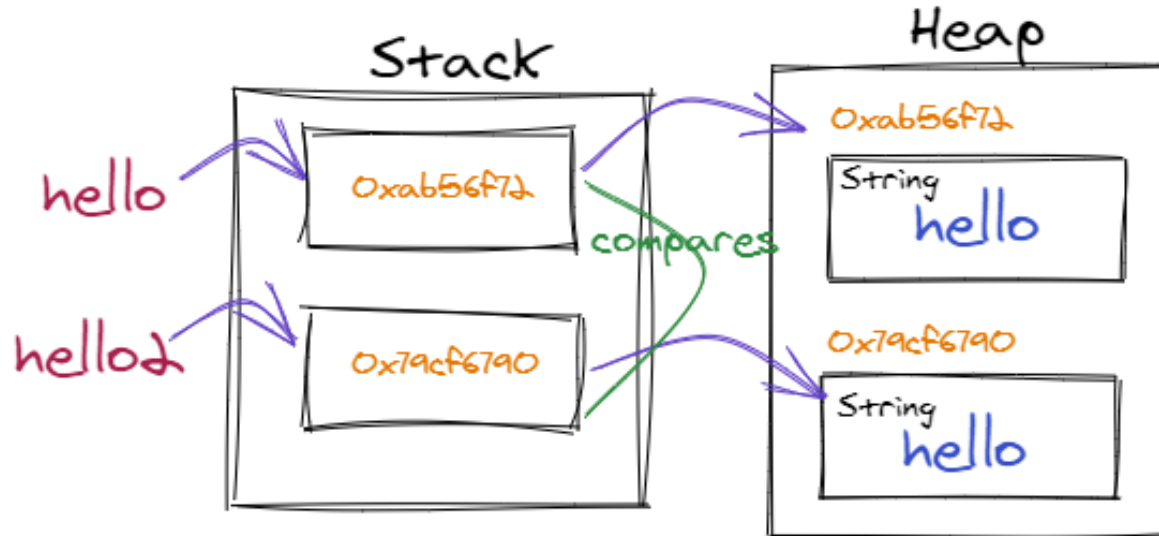
There are 2 Types of Equality in Java

1. Reference Equality (`==`)
2. Value Equality (`.equals()`)

Reference Equality (==)

The equality operator compares the value of 2 variables on the **stack**.

For objects, this is the reference to the Object on the heap

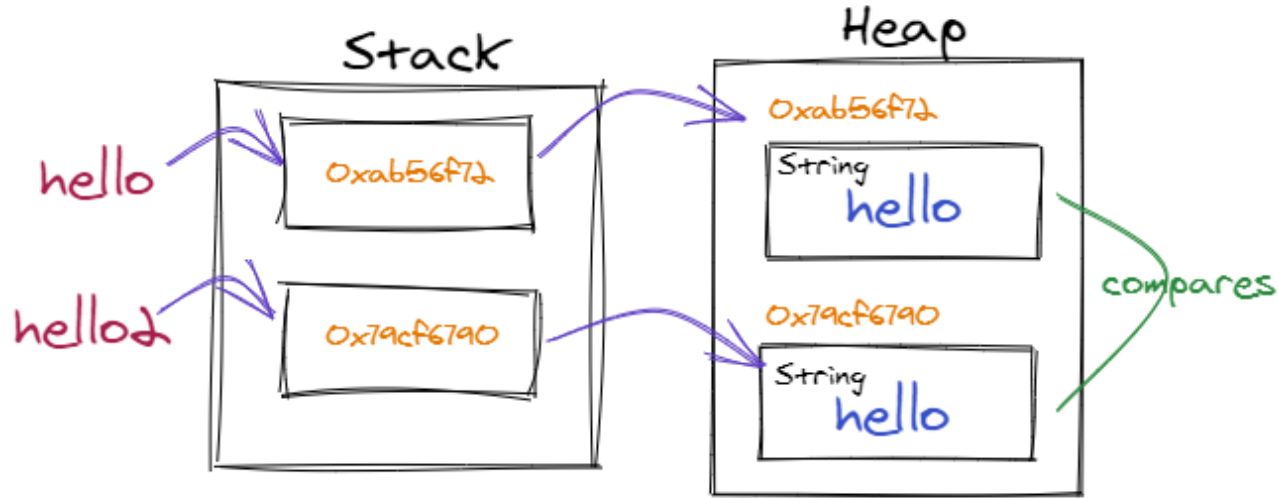


`hello == hello2` **FALSE**

(`0xab56f72` is equal to `0x79cf6790`)

Value Equality (.equals())

The .equals() method compares the value of Objects on the **Heap**



`hello.equals(hello2)` **TRUE**
("hello" is equal to "hello")

String methods

Objects contain behaviors represented by methods. **Methods can be invoked on an object by calling the method with the dot (.) operator and must be ended using parentheses ()**.

The String class contains many methods to manipulate strings. *Since String is immutable, these methods do not change the original String, but create a new one that must be assigned to a variable.*

```
String name = "Rachelle";
```

```
String name = name.toUpperCase();
```

```
name → "RACHELLE"
```

Characters in a String

Characters in a String are internally stored in a char array, so many of the methods use the index of the characters in the char array to select a position in the string.

String name = "Tech Elevator";

T	e	c	h		E	l	e	v	a	t	o	r	r
0	1	2	3	4	5	6	7	8	9	10	11	12	13

`name.length()` ← returns 13

`name.charAt(3)` ← returns 'h'

A for loop can be used to access each character in a string:

```
for ( int i = 0 ; i < name.length() ;  
i++ ) {  
    System.out.println(  
name.charAt( i );  
}
```


String.substring()

The String substring() method returns a portion of a string. It takes 2 arguments, the first is an **inclusive starting index**, and the second is the **exclusive ending index**. So substring() will return a new String with the character starting at the starting index and ending, but not including, the character at the ending index.

```
String name = "Tech Elevator";
```

T	e	c	h		E	l	e	v	a	t	o	r	r
0	1	2	3	4	5	6	7	8	9	10	11	12	13

```
name.substring( 2, 6 ) → returns "ch E"
```

T	e	c	h		E	l	e	v	a	t	o	r	r
0	1	2	3	4	5	6	7	8	9	10	11	12	13

If only a starting index is provided, then a new String will be returned starting at that index through the end of the string.

```
name.substring( 5 ) → returns "Elevator"
```

Common String Methods

Method	Return Type	Description
contains(string)	boolean	True if the String contains the String passed as an argument.
startsWith(string) endsWith(string)	boolean	True if the String starts with or ends with the string passed as an argument
indexOf(string)	int	Returns the starting index of the String passed in the argument
replace(string1, string2)	String	replaces string1 with string2
toLowerCase() toUpperCase()	String	Returns the String in all upper or lower case
split(str)	String[]	Splits the str into an array using the str in the argument as a delimiter
equals(string) equalsIgnoreCase(string)	boolean	True if the value of the string is equal to the string passed in the argument. equalsIgnoreCase() compares the string without case.
trim()	String	removes whitespace from the beginning and ending of the String

Objects Introduction

- In-memory data structure
 - Combines state and behavior into a usable and useful abstraction
- Each object lives in memory and is separate from every other object
 - Reference type
- We don't write objects, we write classes
 - In order to make objects, we must write definition of that object, which is called a class
 - Class is a blueprint
 - Grouping of variables and methods in source code
 - Defines what object will be like when object is created

Objects Introduction

e.g.

```
House houseAt100WestSt = new House(2, 2.5, "Red Brick");
```

- First, declare variable that will hold the object (houseAt100WestSt)
- Then, instantiate a new object from a class
 - Initialize variables with the initial values sent in as arguments (maybe)

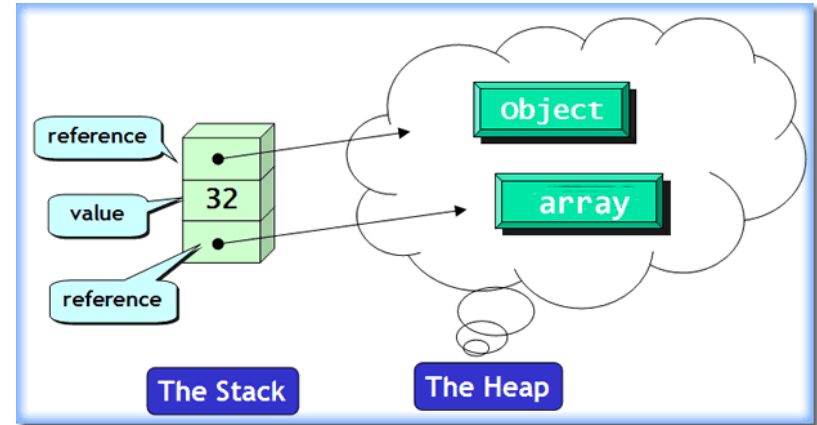
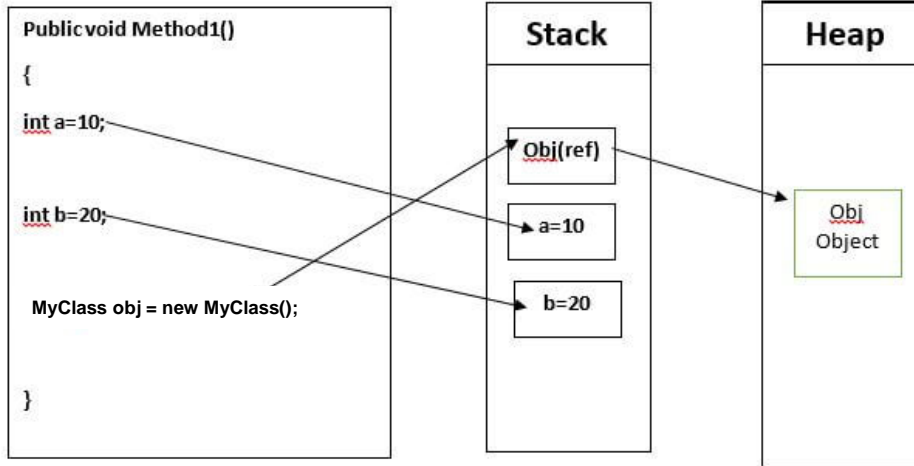
Reference vs Value Types

- You have now encountered various **primitive data types (value types)**: int, double, boolean, float, char, etc.
 - Allocated on Stack
- We will now discuss **reference types**:
 - You have encountered these already - Arrays and Strings are reference types.
 - Objects that you instantiate from classes that you write are also reference types.

```
Car myCar = new Car();
```

- Allocated on Heap

Stack and Heap



- Primitive variables are stored on Stack
- Arrays and objects are stored on Heap

Properties and Methods

Reference types often have properties (also called members, or data members) and methods.



These vehicles were created from the same blueprint. The blueprint specifies that each vehicle should have a color, **color is therefore a property of the object.**

Objects also have methods. Again, consider some of the things a vehicle can do: start the engine, go in reverse, check how much fuel it has left.

Initializing an object

- All objects need to be initialized or instantiated
- (Strings are special and have the ability to be instantiated without new)
- `House myHouse;`
 - Object is declared but not instantiated, value will be null
 - Must instantiate before using

Objects: Arrays

Let's consider Arrays in the context of objects.

- Arrays have a length **property**: `myArray.length`
- Arrays also have **static methods** (just like wrapper class methods):

```
boolean check = Arrays.equals(oneStringArray, twoStringArray);  
System.out.println(check);
```

To access an object's properties or methods we use the dot operator as observed above. Methods have a set of parentheses.

Arrays Methods (just a few)

- `Arrays.toString(intArray)` – allows you to print all values of an array without a loop
- `Arrays.copyOf(intArray, 5)` – creates a new array of 5 elements and copies in (first 5) elements of `intArray`
- `Arrays.equals(oneArray, twoArray)` – returns true if all values in the array are equal to one another
- `Arrays.fill(doubleArray, myDouble)` – assigns the value of `myDouble` to all elements in `doubleArray`
- `Arrays.sort(myArray)` – sorts the array in ascending order

Strings: length method

Unlike arrays, to obtain the length of a string, a method is called. We know this because of the presence of parenthesis.

```
String myString = "Pure Michigan";  
int myStringLength = myString.length();  
System.out.println(myStringLength);  
// The output is 13.
```

- Note that no parameters were taken, nothing goes inside the parenthesis.
- The method's return is an integer, we can assign it to an integer if needed.

Strings: charAt method

The charAt method for a string returns the character at a given index. The index on a String is similar to that of an Array, namely that it starts at zero.

```
String myString = "Pure Michigan";  
char myChar = myString.charAt(1);  
System.out.println(myChar);  
// The output is u.
```

- Note that charAt takes 1 parameter, the index number indicating the position in the String you want to extract.
- The method's return value is of type char.

Strings: indexOf method

The indexOf method returns the starting position of a character or String.

```
String myString = "Pure Michigan";  
int position = myString.indexOf('u');  
int anotherPosition = myString.indexOf("Mi");  
  
System.out.println(position); // 1  
System.out.println(anotherPosition); // 5
```

- Note that indexOf takes one parameter, what you're searching for.
- The method's return is an integer, if nothing is found it will return a -1. If there are multiple matches, it will return the index corresponding the first one.

Strings: substring method

The substring method returns part of a larger string.

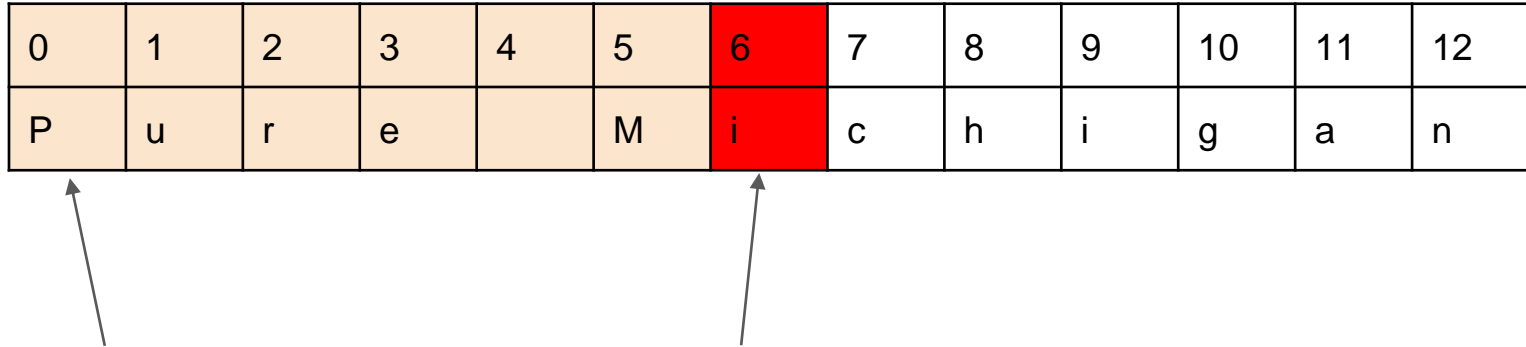
```
String myString = "Pure Michigan";  
String mySubString = myString.substring(0, 6);  
System.out.println(mySubString);  
// output: Pure M
```

- Substring requires two parameters, the first is the starting point. The second parameter is a non-inclusive end point (more on this on the next slide).
- It returns a String, so you can assign the output to a String.

Strings: substring method

Just like with arrays, drawing a table of elements or position is a great way to visualize these concepts. Consider the following method call `substring(0, 6)`

0	1	2	3	4	5	6	7	8	9	10	11	12
P	u	r	e		M	i	c	h	i	g	a	n



The first parameter is 0, denoting we will start the new String from the 0th position.

The second parameter is the stopping point. The stopping point (6th element) is not included in the final String.

Hence, the output from the previous page is:
Pure M

Strings: Comparisons

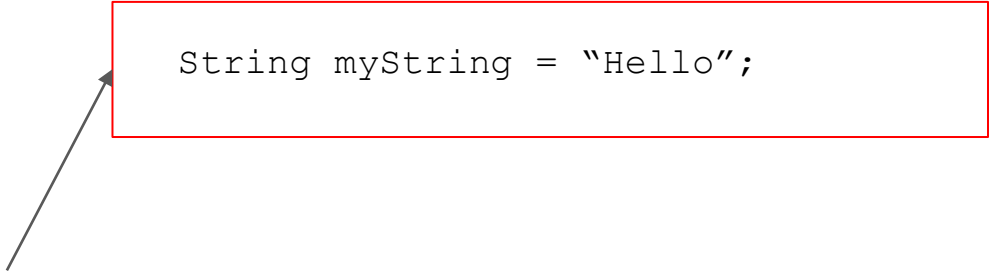
The proper way to compare Strings is to use the equals() method.

```
String myString = "Pure Michigan";  
String myOtherString = "Pure Michigan";  
String yetAnotherString = "Ohio so much  
to discover";  
  
if (myString.equals(myOtherString)) {  
    System.out.println("match");  
}
```

Do not use == to compare Strings!

Objects: References

The previous discussion on why `==` should not be used with Strings illustrates an important concept concerning assigning objects (like Strings and Arrays) to variables.



```
String myString = "Hello";
```

myString is a
reference to
where "Hello" is
stored in memory
(in the heap).

A reference does not actually store an object, it only tells you where it is in memory.

The “new” keyword

- Java is built around thousands of “blueprints” called classes and provides you with the ability to create your own classes.
- The **new** keyword is typically used to create an instance of a class.
- We refer to these instances as **objects** of a specific class.
- We have already seen this before, consider the declaration of an array.

```
int [] scores = new int[5];
```

- If we use the concepts we just learned, the above statement means, create a reference (key) of type integer array. Proceed to create a new instance of this array of length 5.
- Strings aren’t required to follow this convention... but if you want to you can:

```
String aString = new String("Hello");
```