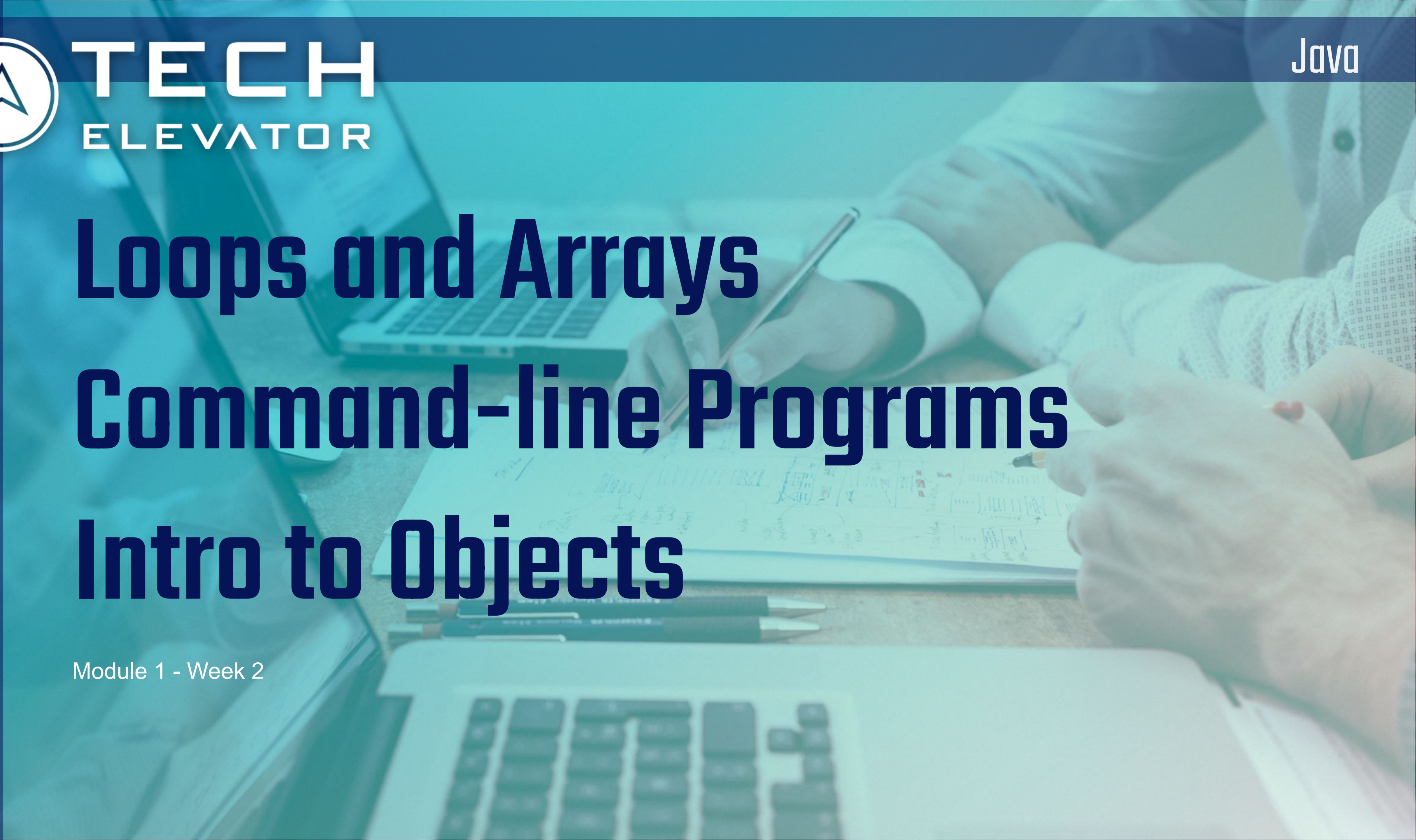
Java

Loops and Arrays
Command-line Programs
Intro to Objects

Module 1 - Week 2

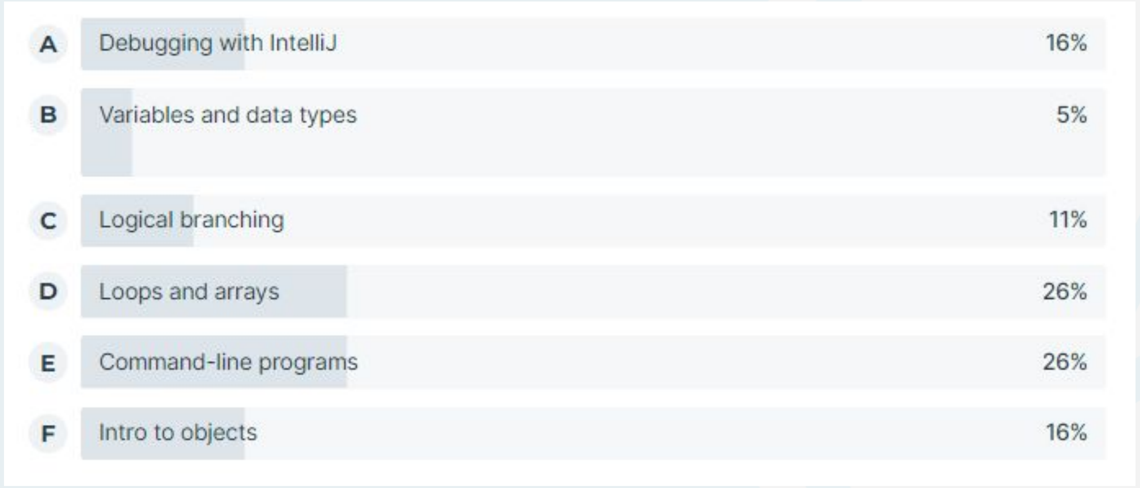


Interview Question:
When would you want to use a while loop instead of a for loop?

Housekeeping

Asking for help...

- Office Hours! We cover homework.
- Post in **#nlr-pt-5-java-blue**.
- Add ticket emoji 🎫 reaction to any slack message to create a help ticket. More details here.
- Schedule 1:1 with me.
- DM me on Slack.



A	Debugging with IntelliJ	16%
B	Variables and data types	5%
C	Logical branching	11%
D	Loops and arrays	26%
E	Command-line programs	26%
F	Intro to objects	16%

Schedule

Week	Topics
Week 0	Welcome, Intro to Tools
Week 1	Variables and Data Types, Logical Branching
Week 2	Loops and Arrays, Command-Line Programs, Intro to Objects
Week 3	Collections
Week 4	**Mid-Module Project**
Week 5	Classes and Encapsulation
Week 6	Inheritance, Polymorphism
Week 7	Unit Testing, Exceptions and Error Handling
Week 8	File I/O Reading and Writing
Week 9	**End-of-module project**
Week 10	**Assessment**

Objectives

- Learning objectives
  - **Use for loops for iterative logic including sequentially processing values in an array**
  - Use methods of the String class for text processing and manipulation
  - Accept user input from `stdin`

Arrays

- Arrays are a collection of elements that all have the same data type.
- Examples:
  - The 10 day weather (temps)

90.0	86.2	88.3	82.2	80.1	86.4	80.7	78.7	82.8	86.1

```
double[] temps = new double[10];
```

  - In football, the points per quarter

3	7	14	3

```
int[] team1Score = new int[4];
```

  - A roster of names

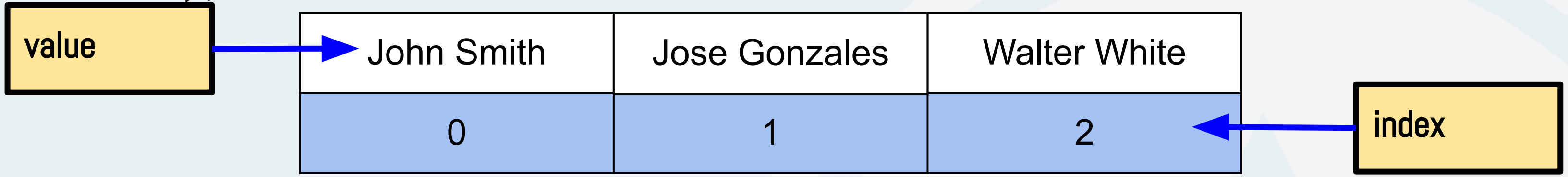
John Smith	Jane Doe	Walter White

```
String[] names = {"John Smith","Jane Doe","Walter White"};
```

Array Index

Let's look at the array of strings example:

```
String[] names = {"John Smith","Jose Gonzales","Walter White"};
```



- You can think of an array as a **series of slots** that hold **data of the same data type** (e.g. **Strings**).
- You can refer to each element in the array by an **index**.
- The **index** will be an **int** value (starting at 0) which increases with each slot.

```
names[1]= "Jose Gonzales";

System.out.println("Name: " + names [1]);
```

For Loops

- The **`for`** statement provides a compact way to iterate over a range of values.
- Repeatedly loops until a particular condition is satisfied.
- The general form of the **`for`** statement can be expressed as follows:

```
for (initialization; termination; increment){
    statement(s)
}
```

- The initialization expression initializes the loop. It's only executed once.
- The statement(s) are executed repeatedly until the loop terminates.
- When the termination expression evaluates to false, the loop terminates.
- The increment expression is invoked after each iteration through the loop. It is typical (although not required) for this to increment or decrement a value.

```
for (int i = 0; i < 5; i++){
    System.out.println("Number is " + i);
}
```

While Loops

- While loops continue looping until a condition is no longer true.
- The general form of the `while` statement can be expressed as follows:

```
while (condition) {
       statement(s)
}
```

- If condition is not met on initial run, the code in the loop will never execute.
- If your condition is based on an index, you must increase or decrease the index manually.

```
int i = 0;

while (i < 5) {
    System.out.println("Number is " + i);
    i++;
}
```

Do-While Loops

- Do-while loops continue looping until a condition is no longer true.
- The general form of the `do-while` statement can be expressed as follows:

```
do {
    statement(s)
}while(condition);
```

- Guaranteed to execute at least once
- If your condition is based on an index, you must increase or decrease the index manually.

```
int i = 0;

do {
    System.out.println("Number is " + i);
    i++;
} while (i < 5);
```

Iterating Arrays

- We can use a loop to sequentially iterate through each element of the array.
- The value of **`team1Score.length`** will be **4**.
- Note that **`i`** is also used to specify the current index of the array so that each element of the array can be printed.

```
int [] team1Score = new int [4];

team1Score[0]= 20;
team1Score[1]= 14;
team1Score[2]= 18;
team1Score[3]= 23;

for (int i = 0; i < team1Score.length; i++) {
System.out.println(team1Score[i]);
}
```

Increment & Decrement Operators

- The increment (**++**) and decrement operator (**--**) increases or decreases a number by 1 respectively.
- You have seen this in the context of a for loop.
- If the operator is behind a variable it is a postfix operator (e.g. **x++**). A variable with a postfix operator is evaluated first, then incremented.
- If the operator is in front a variable it is a prefix operator (e.g. **++x**). A variable with a prefix operator is incremented first, then evaluated.

```
int x = 3;
System.out.println(++x + 4);
System.out.println(x);
```

With a prefix operator, **x** is increased prior to evaluating the addition expression. The values **8** and **4** are printed out.

```
int x = 3;
System.out.println(x++ + 4);
System.out.println(x);
```

With a postfix operator, **x** is increased after evaluating the addition expression. The value **7** and **4** are printed out.

For-each Loops

- The syntax of the Java for-each loop is:

```
for (type variableName : arrayName) {
  // code block to be executed
}
```

- arrayName - an array or a collection
- variableName - each item of array/collection is assigned to this variable
- type - the data type of the array/collection

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String car : cars) {
  System.out.println(car);
}
```

Use for loops for iterative logic - Exercise 1

Common difficulties

- thinking loops must always involve an array
- stopping before going out of bounds when iterating through an array
- confusing the iterator variable with the array element at that index

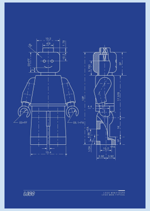
Objectives

- Learning objectives
  - Use for loops for iterative logic including sequentially processing values in an array
  - **Use methods of the String class for text processing and manipulation**
  - Accept user input from `stdin`

Classes & Objects

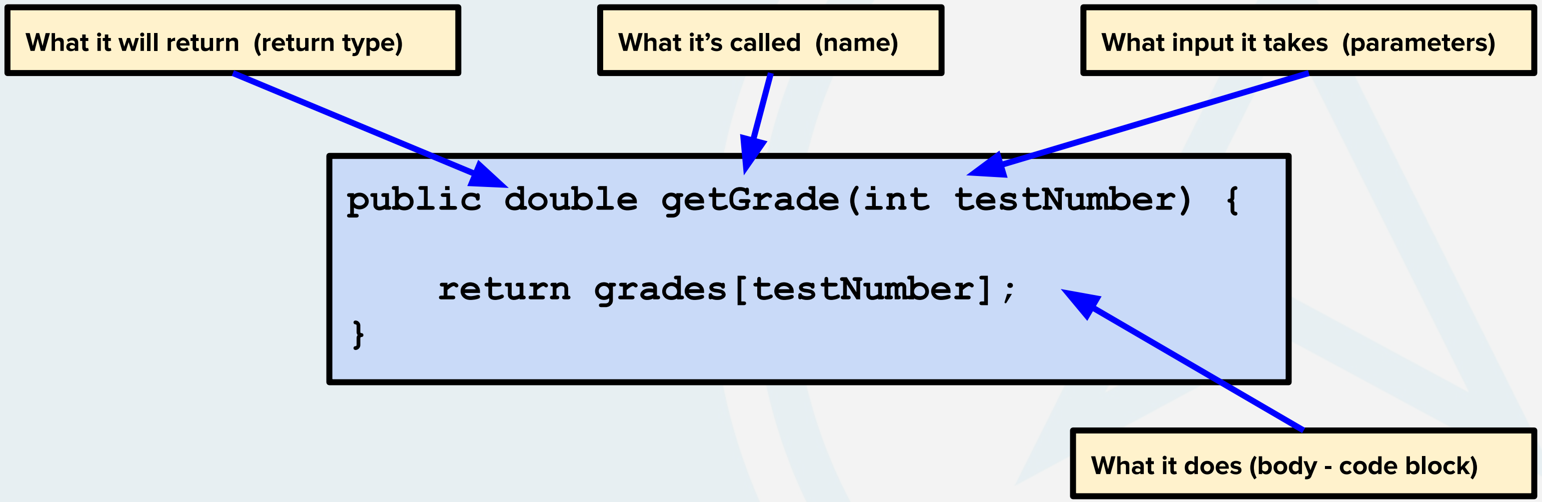
- A **`Class`** is source code; a grouping of variables and methods, used to create objects.
- An **object** is an in-memory instantiation of a Class.

The blueprint on the left was used to build the Lego men on the right. The blueprint specifies that the men's overalls will have a color but the actual color can be different for each man.

Class 	Object
A template for creating (***instantiating***) objects within a program	An ***instance*** 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

Methods

- Methods are related statements that complete a specific task or set of tasks.
- Methods can be called from different places in the code.
- When called, inputs can be provided to a method.
- Methods can also return a value to its caller.
- Methods are Java's versions of functions. You can think of this as a process that could potentially take several inputs and use it to generate output.


What it will return (return type)
What it's called (name)
What input it takes (parameters)
What it does (body - code block)

```
public double getGrade(int testNumber) {

    return grades[testNumber];
}
```

Properties & Methods

- Some classes are "built in" to Java like **`String`**, **arrays** and the **wrapper classes.**
- We can also define our own classes.
- Classes define properties (aka member variables) and methods.
  - Properties are attributes that define an object's state.
  - Methods define an object's behaviors.

```
public class Student {
    private String name;
    private double[] grades = new double[10];

--------------------------------------------------------------------


    public String getName() {
        return name;
    }

    public void addGrade(int testNumber, double grade) {
        grades[testNumber] = grade;
    }

    public double getGrade(int testNumber) {
        return grades[testNumber];
    }
 ...
}
```

properties

methods

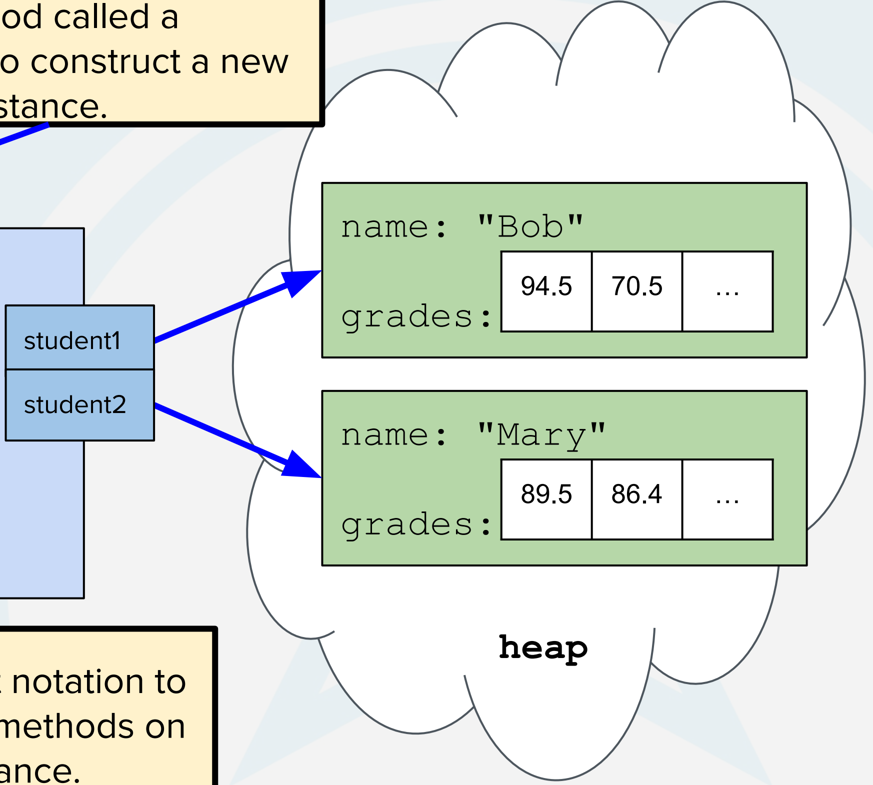
Instances

Declare a new variable of type **`Student`** and assign to it a new instance (object) of the **`Student`** class.

Use the `new` keyword with a special method called a constructor to construct a new `Student` instance.

```
Student student1 = new Student("Bob", 1);
student1.addGrade(1, 94.5);
student1.addGrade(2, 70.5);


Student student2 = new Student("Mary", 2);
student2.addGrade(1, 89.5);
student2.addGrade(2, 86.4);
```



Use dot notation to invoke methods on the instance.

Wrapper Classes

- All primitive data types have powerful non-primitive equivalents (called **wrapper classes**).
- **Wrapper classes** provide a way to use primitive data types (int, boolean, etc..) as objects.
- In most cases, wrappers and primitives can be used interchangeably without explicit conversion (boxing & unboxing).
- Use wrapper classes when…
  - Required when using Collections or Generics (more about this later)
  - If you want a variable that can be null or return from a method that can return null
  - Take advantage of methods, attributes or constants defined on the wrapper class (e.g. **`Integer.MAX_SIZE`**, **`Boolean.parseBoolean(string)`**)

Primitive	**Wrapper**	**Example**
`int`	`Integer`	`Integer number = 3;`
`double`	`Double`	`Double amount = 3.14;`
`boolean`	`Boolean`	`Boolean done = false;`

Parsing Strings

There are times when we need to take a String value and convert it into a different data type.

```
System.out.print("Please enter your height in inches: ");
String heightInput = scanner.nextLine();
```

Parse methods that can parse a String to that data type are available for each of the basic data types using their wrapper class.   The String must contain characters that are valid for the data type it is being parsed into.

Primitive	Wrapper	Parse Method
`int`	`Integer`	`Integer.parseInt(string)`
`long`	`Long`	`Long.parseLong(string)`
`double`	`Double`	`Double.parseDouble(string)`
`boolean`	`Boolean`	`Boolean.parseBoolean(string)`

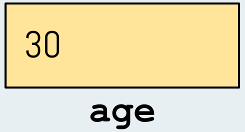
Primitive vs Reference Variables

- The basic difference is that primitive variables store the actual values, whereas reference variables store the addresses of the objects they refer to.
- `String` and user defined classes are reference types.

Primitive

int age = 25;

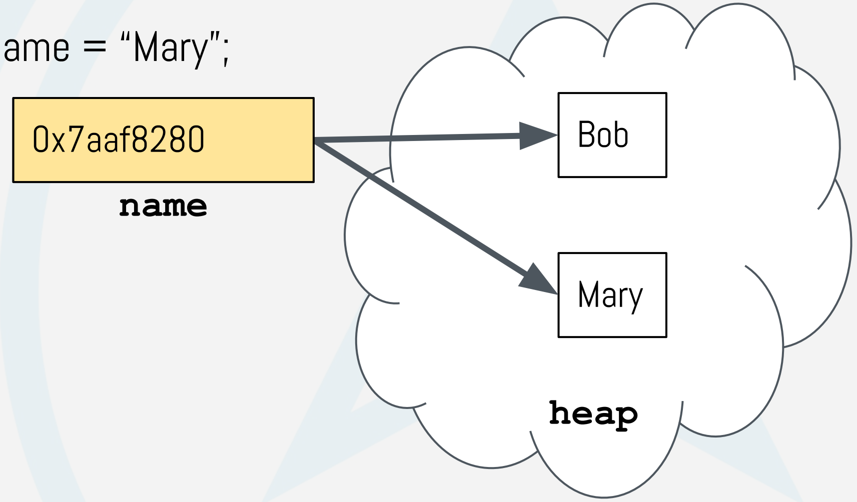
age = 30;



Reference

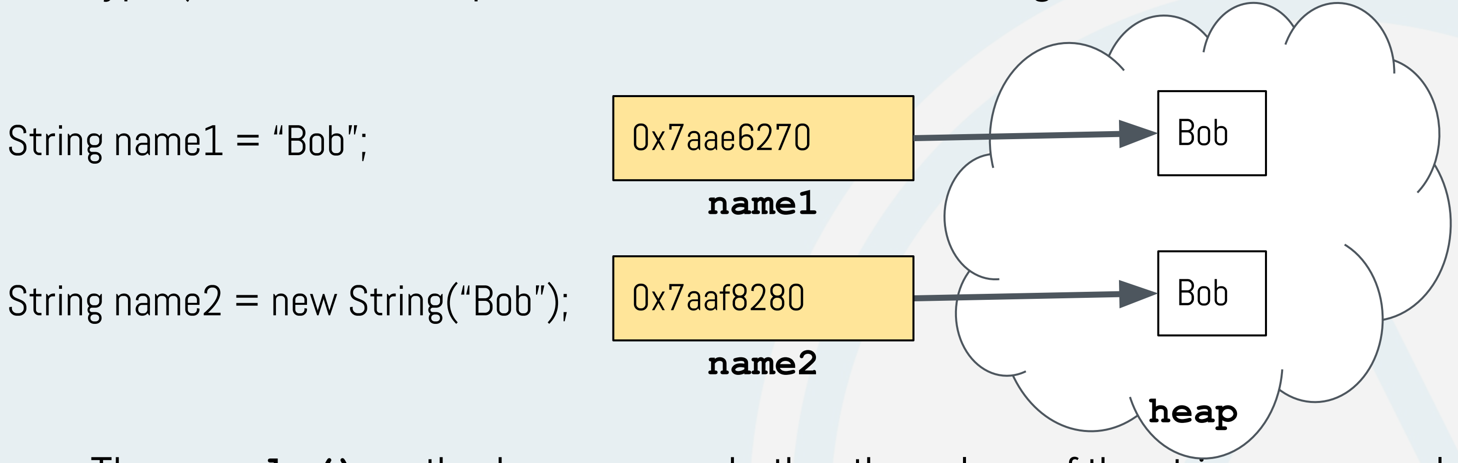
String name = “Bob”;

name = “Mary”;



Comparing Strings & Reference Types

- You should not use the equality operator == to compare `Strings` (and other reference types) because it compares the references of the strings, not the values.



- The **`equals()`** method compares whether the values of the strings are equal.

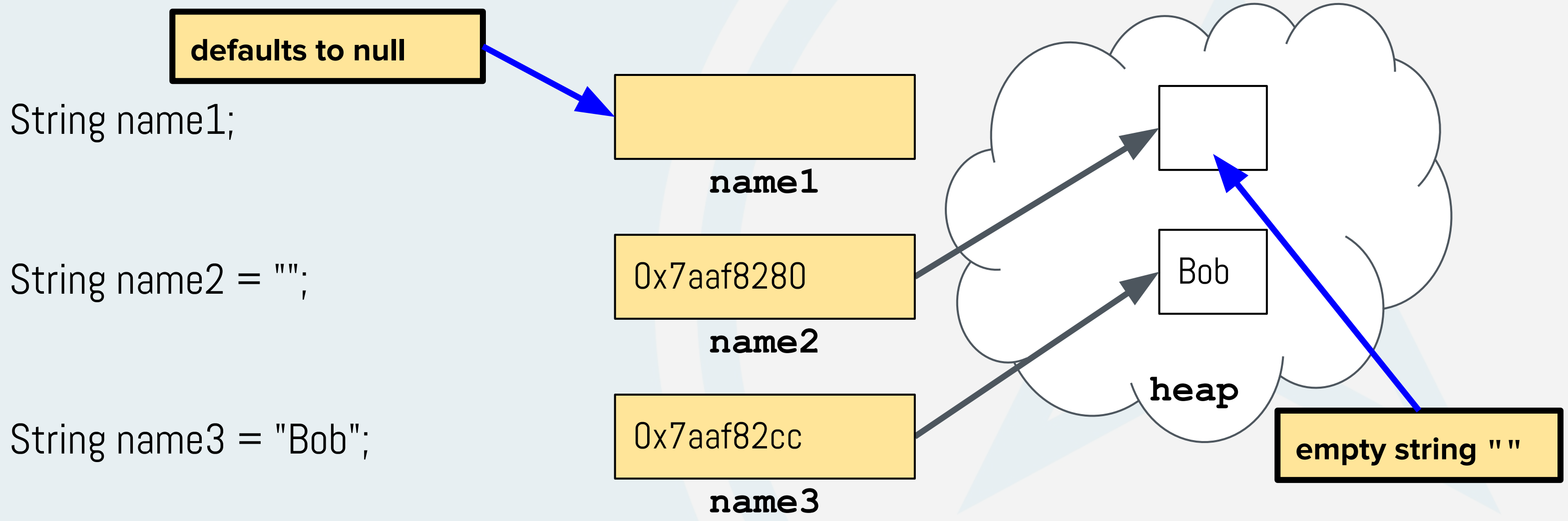
```
if (name1.equals(name2)) {
     System.out.println("The strings are equal!);
}
```

Don't use == here!

Use `equals()` instead!

Null

- When variables for **Reference Types** are created they default to **`null`**. They are **`null`** until they are assigned a reference to an instantiated object.
- **`null`** `String` is not the same thing as empty `String`. **`null`** means no value. Empty refers to an empty `String` with a value of `""`.
- A **`NullPointerException`** occurs when trying to use a method or property on an object when the variable does not contain a reference and is **`null`**.

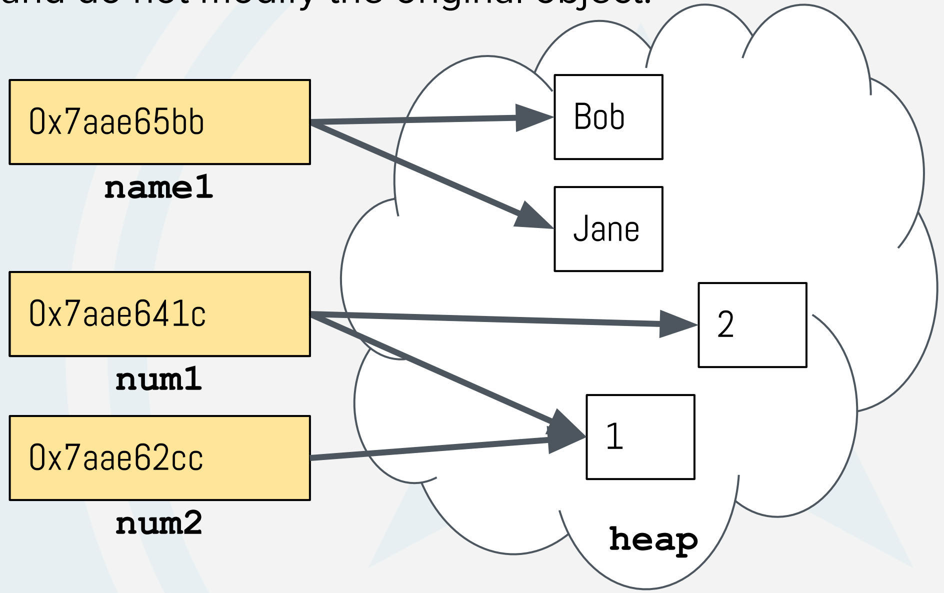


Immutable Objects

- An object whose state cannot be changed after it is created is called **immutable**.
- `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.
- All primitive wrapper classes are immutable, so operations like addition and subtraction create a new object and do not modify the original object.

```
String name1 = "Bob";
name1 = "Jane";

Integer num1 = 1;
Integer num2 = num1;
num1++;
```



String Methods

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.

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

```
String name = "Tech Elevator";

name.length();  // returns 13
name.charAt(3); // returns 'h'

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

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

String Methods

Like most classes, the `String` class defines methods. Here are some common `String` methods:

Method	Return Type	Description
`contains( string )`	`boolean`	True if this string contains the string passed as an argument.
`charAt( index )`	`char`	Returns the character at a given index of this string.

`startWith( string)`
`endsWith( string )`
	`boolean`	True if this string starts with or ends with the string passed as an argument.

`indexOf( string )`	`int`	Returns the starting index in this string of the string passed in the argument.
`replace( string1, string2)`	`String`	Replaces string1 with string2 in this string.

`toLowerCase()`
`toUpperCase()`
	`String`	Returns a copy of this string in all upper or lower case

`split( str )`	`String[]`	Splits this string into an array using the str in the argument as a delimiter

`equals( string )`
`equalsIgnoreCase( string )`
	`boolean`	True if the value of this string is equal to the string passed in the argument. equalsIgnoreCase() compares the string without case.

`trim()`	`String`	Returns a copy of this string without whitespace at the start and ending.

String Substring

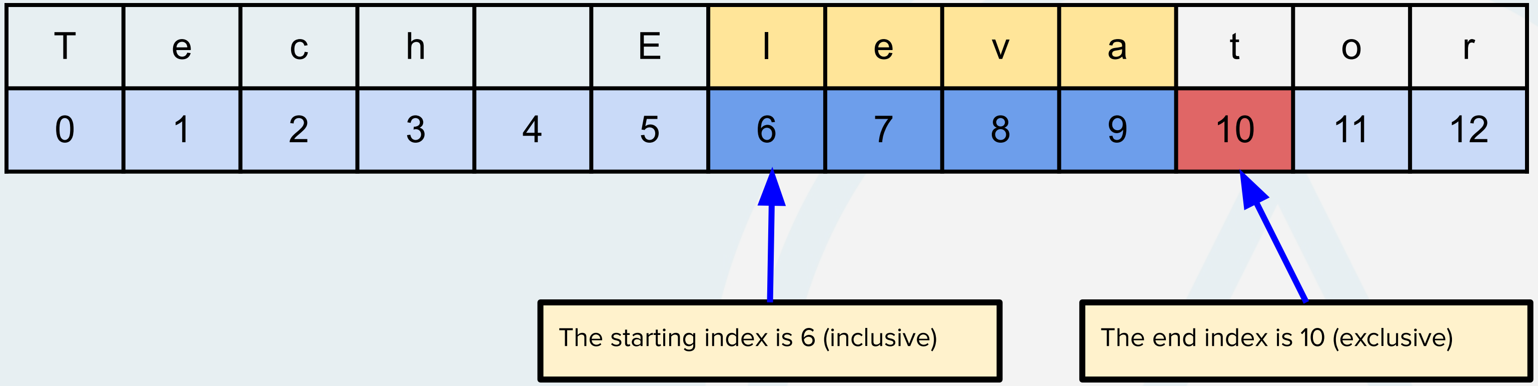
The `substring()` method for a `String` returns part of a larger string.

- There are two versions of `substring()`.
  - The first version requires two parameters. The integer first parameter is the starting index. The second integer parameter is a non-inclusive end index.
  - The second version requires one parameter - the integer starting index. The end point is the end of the string.
- It returns a `String`, so you can assign the output to a `String` variable.

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

String mySubString2 = myString.substring(5);
System.out.println(mySubString2); // output: Michigan
```

String Substring



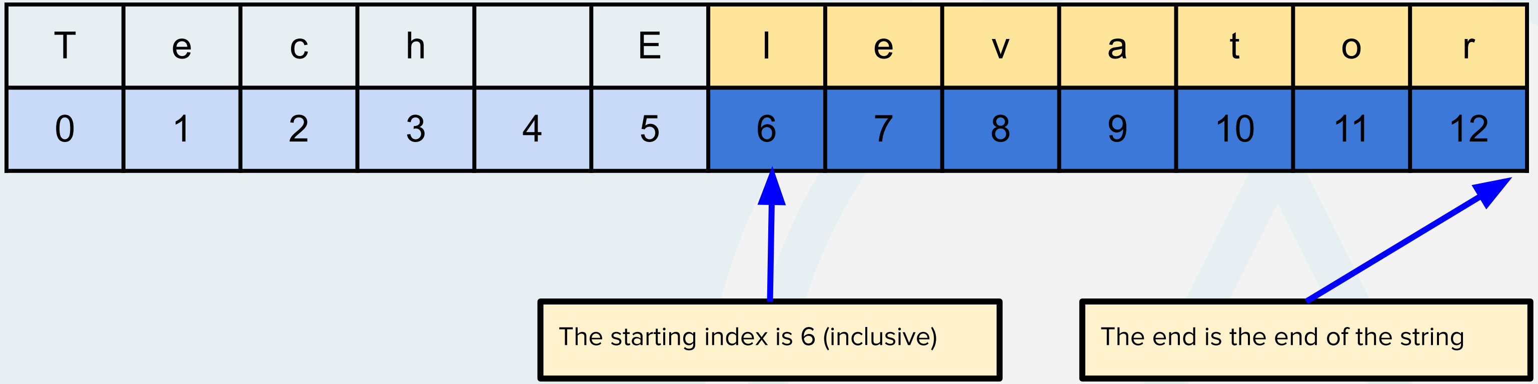
T	e	c	h		E	l	e	v	a	t	o	r
0	1	2	3	4	5	6	7	8	9	10	11	12

The starting index is 6 (inclusive)

The end index is 10 (exclusive)

```
String myString = "Tech Elevator";
String mySubString = myString.substring(6, 10);
System.out.println(mySubString); // output: leva
```

String Substring


T
e
c
h
E
l
e
v
a
t
o
r
0
1
2
3
4
5
6
7
8
9
10
11
12
The starting index is 6 (inclusive)
The end is the end of the string

```
String myString = "Tech Elevator";
String mySubString = myString.substring(6);
System.out.println(mySubString); // output: levator
```

Use methods of the String class - Exercise 2

Common difficulties

- confusing char primitives with single character strings
- understanding the difference between an empty string and null
- remembering to compare strings with the equals() method instead of ==
- determining which indices to use with substring()
- remembering that invoking methods on strings won't change their state due to their immutability

Objectives

- Learning objectives
  - Use for loops for iterative logic including sequentially processing values in an array
  - Use methods of the String class for text processing and manipulation
  - **Accept user input from `stdin`**

System.in & System.out

- **`System.in`** is the standard input stream and refers to the keyboard
- To read from the keyboard, we need to create a Scanner object

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

Note the **`new`** keyword!

- **`scanner.nextLine()`** gets text from the input stream up until a newline (the user presses Enter). The text is returned as a String and the newline is disregarded.

```
String userInput = scanner.nextLine();
```

- **`System.out`** is the standard output stream and refers to the console (monitor/terminal)
- **`System.out`** is a PrintStream to which you can write characters.
  - Normally outputs the data you write to it to the CLI console / terminal.
  - Often used from CLI programs to display the result to the user.
  - Also often used to print debug statements from a program.

*See formatting rules: https://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html

Command Line

Command line program to read in user name and height in inches, convert the height to centimeters and output to console.

```java
import java.util.Scanner;

public class InputReader {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Please enter your name: ");
        String nameInput = scanner.nextLine();

        System.out.print("Please enter your height in inches: ");
        String heightInput = scanner.nextLine();
        double heightInches = Double.parseDouble(heightInput);
        double heightCms = heightInches * 2.54;

        System.out.println("Hi " + nameInput + ". Your height is  " +
            heightCms + " centimeters.");
    }
}
```

To use the scanner object, we must import in the correct class.

Create an object of type **`Scanner`**

The input is read and stored into a **`String`** called **`nameInput`**.

The input is read and stored into a String called **`heightInput.`**

`heightInput` is converted into an **`double`** using the **`Double`** wrapper class.

Use methods of the String class - Exercise 3

Common difficulties

- confusing char primitives with single character strings
- understanding the difference between an empty string and null
- remembering to compare strings with the equals() method instead of ==
- determining which indices to use with substring()
- remembering that invoking methods on strings won't change their state due to their immutability

Objectives

- Learning objectives
  - Use for loops for iterative logic including sequentially processing values in an array
  - Use methods of the String class for text processing and manipulation
  - Accept user input from `stdin`