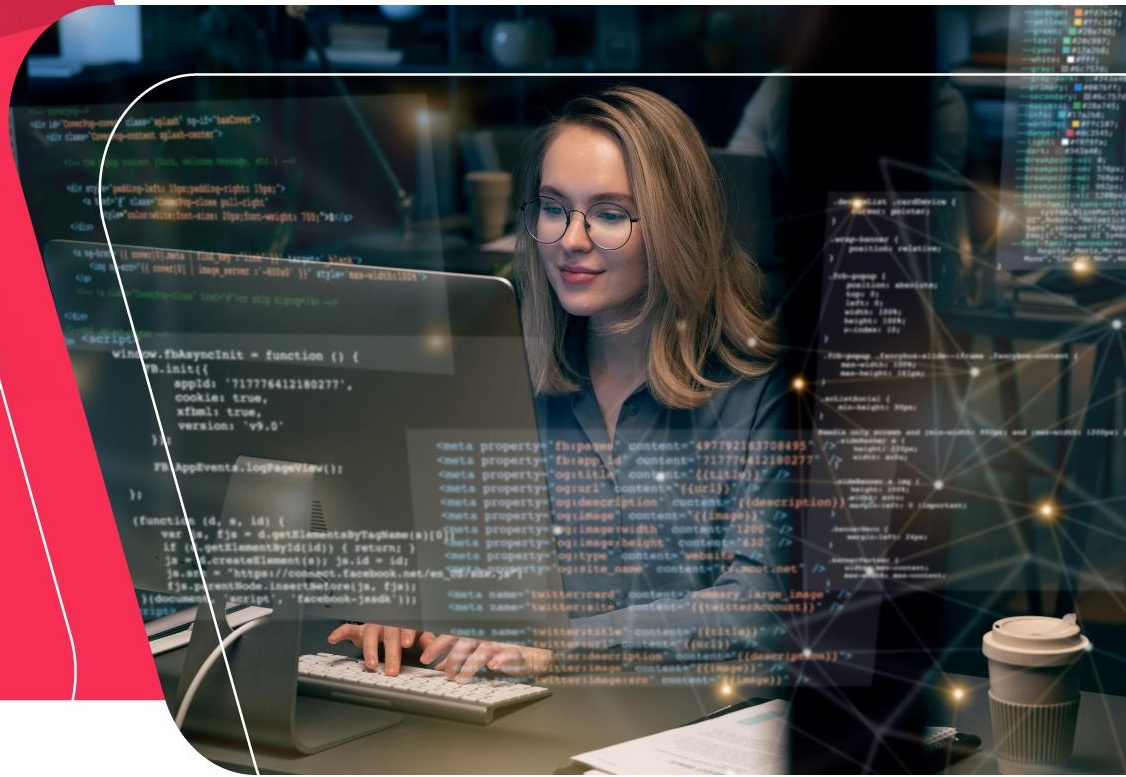
TIA-Academy

Java Bootcamp

Day 11



Technologies will be Use

- JDK 8/**11**/15
- JRE 8/**11**/15
- Writing Code using Notepad++ (For Windows) or Vim (For Linux and Mac)
- Compiling with Command Prompt (For Windows) or Terminal (For Linux and Mac)
- ***Forbidden of using IDE Based Development Tools as of Intellij IDEA or Eclipse***

TIA-Academy

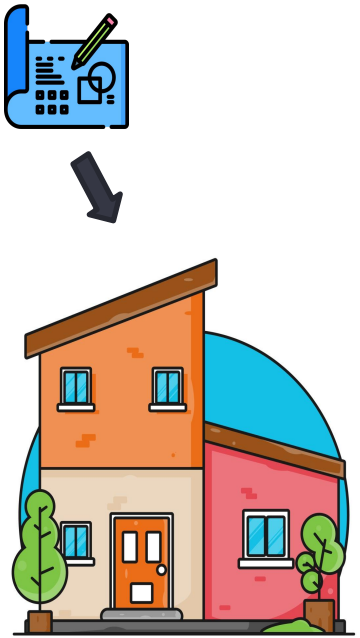
Java Class and Objects

Java Class and Objects

- Java is an object-oriented programming language. **The core concept of the object-oriented approach is to break complex problems into smaller objects.**
- An object is any **entity** that has a **state** and **behavior**. For example, a bicycle is an object. It has
  - **States**: idle, first gear, etc
  - **Behaviors**: braking, accelerating, etc.
- Before we learn about objects, let's first know about classes in Java.

Java Class

- **A class is a blueprint for the object**. Before we create an object, we first need to define the class.
- We can think of the **class as a sketch (prototype) of a house**. It contains **all the details about the floors, doors, windows, etc**. Based on these descriptions we build the house. **House is the object**.
- Since many houses can be made from the same description, we can create many objects from a class.



Create a class in Java

- We can create a class in Java using the class keyword. For example,

```
class ClassName {
  // fields
  // methods
}
```

- Here, **fields (variables)** and **methods** represent the state and behavior of the object respectively.
  - fields are used to **store data**
  - methods are used to **perform some operations**

Create a class in Java

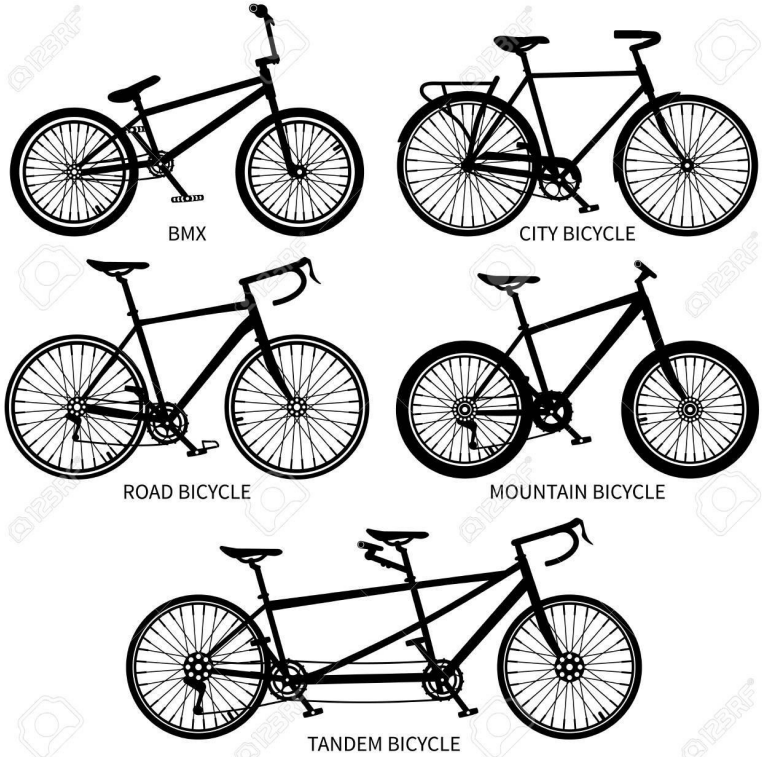
- For our bicycle object, we can create the class as

```
class Bicycle {
  // state or field
  private int gear = 5;

  // behavior or method
  public void braking() {
    System.out.println("Working of Braking");
  }
}
```

- In the above example, we have created a class named Bicycle. It contains a field named gear and a method named braking().
- Here, **Bicycle is a prototype**. Now, we can **create any number of bicycles using the prototype**. And**, all the bicycles will share the fields and methods of the prototype**.

Java Objects

- An object is called an instance of a class.
- For example, suppose **Bicycle is a class** then BMX, MountainBicycle, CityBicycle, RoadBicycle, TouringBicycle, etc can be considered as objects of the class.



Create a Object in Java

- Here is how we can create an object of a class.

```
className object = new className();

// for Bicycle class
Bicycle sportsBicycle = new Bicycle();

Bicycle touringBicycle = new Bicycle();
```

- We have used the **new** keyword along with the **constructor** of the class to create an object. **Constructors are similar to methods and have the same name as the class.** For example, **Bicycle()** is the constructor of the Bicycle class.

Create a Object in Java

- Here, **sportsBicycle** and **touringBicycle** are the names of objects. We can use them to access fields and methods of the class.
- As you can see**, we have created two objects of the class**. We can **create multiple objects of a single class in Java**.
- Note: ***Fields and methods of a class are also called members of the class.***

Access Members of a Class

- We can use the name of objects along with **the . operator** to access members of a class. For example,
- In the example, we have created a class named Bicycle.
- It **includes a field named gear and a method named braking().**

```
class Bicycle {

  // field of class
  int gear = 5;

  // method of class
  void braking() {
    ...
  }
}

// create object
Bicycle sportsBicycle = new Bicycle();

// access field and method
sportsBicycle.gear;
sportsBicycle.braking();
```

Access Members of a Class

- Notice the statement,

```
Bicycle sportsBicycle = new Bicycle();
```

- Here, we have created an object of Bicycle named sportsBicycle. We **then use the object to access the field and method** of the class.
  - `sportsBicycle.gear` - access the field gear
  - `sportsBicycle.braking()` - access the method `braking()`

- We have mentioned the word **method** quite a few times. You will learn about Java methods in detail in the next chapter.

Example: Java Class and Objects

```
class Lamp {

  // stores the value for light
  // true if light is on
  // false if light is off
  boolean isOn;

  // method to turn on the light
  void turnOn() {
    isOn = true;
    System.out.println("Light on? " + isOn);

  }

  // method to turnoff the light
  void turnOff() {
    isOn = false;
    System.out.println("Light on? " + isOn);
  }
}
```

Example: Java Class and Objects

```
class Main {
  public static void main(String[] args) {

    // create objects led and halogen
    Lamp led = new Lamp();
    Lamp halogen = new Lamp();

    // turn on the light by
    // calling method turnOn()
    led.turnOn();

    // turn off the light by
    // calling method turnOff()
    halogen.turnOff();
  }
}
```

Example: Java Class and Objects

- Output:

```
Light on? true
Light on? false
```

- In the above program, we have created a class named **Lamp**.
- It contains a variable: **isOn** and two methods: **turnOn()** and **turnOff().**

Example: Java Class and Objects

- Inside the Main class, we have created two objects: **led** and **halogen** of the **Lamp** class.

  We then **used the objects to call the methods** of the class.

  - led.turnOn() - It sets the **isOn** variable to **true** and **prints the output**.
  - halogen.turnOff() - It sets the **isOn** variable to **false** and **prints the output**.
- The variable **isOn defined inside the class is also called an instance variable**. It is because when we create an object of the class, it is called an instance of the class. And**, each instance will have its own copy of the variable.**
- That is, **led and halogen objects will have their own copy of the isOn variable.**

Example: Create objects inside the same

- Note that in the previous example, we have created objects inside another class and accessed the members from that class.
- However, we can also create objects inside the same class.
- Output:

```
Light on? true
```

- Here, we are creating the object inside the main() method of the same class.

```
class Lamp {

  // stores the value for light
  // true if light is on
  // false if light is off
  boolean isOn;

  // method to turn on the light
  void turnOn() {
    isOn = true;
    System.out.println("Light on? " + isOn);

  }

  public static void main(String[] args) {

    // create an object of Lamp
    Lamp led = new Lamp();

    // access method using object
    led.turnOn();
  }
}
```

TIA-Academy

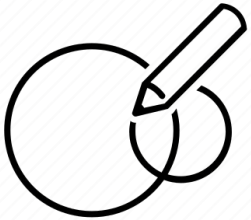
ASSIGNMENT 01

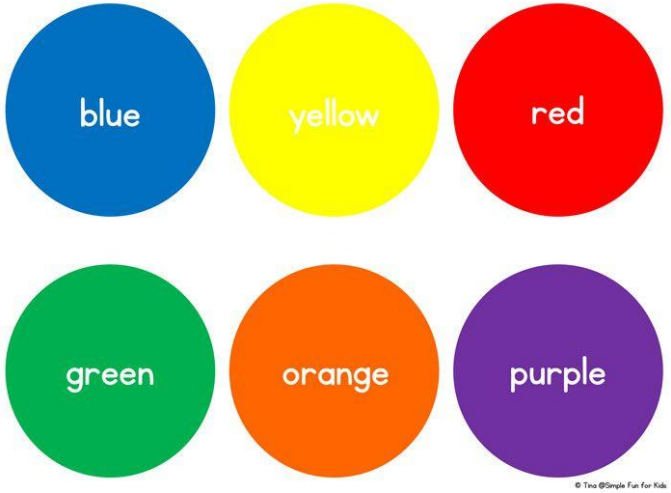
TIA-Academy

Java Methods

Java Methods

- A method is a block of code that performs a specific task.
- Suppose you **need to create a program to create a circle and color it**. You can create two methods to solve this problem:
  - a method to draw the circle
  - a method to color the circle





Java Methods

- **Dividing a complex problem into smaller chunks** makes your program easy to understand and reusable.
- In Java, there are two types of methods:
  - **User-defined Methods**: We can create our own method based on our requirements.
  - **Standard Library Methods**: These are built-in methods in Java that are available to use.
- Let's first learn about user-defined methods.

Declaring a Java Method

- The syntax to declare a method is:

```
returnType methodName() {
  // method body
}
```

- **returnType** - It specifies what type of value a method returns For example if a method has an **int** return type then it **returns an integer value.**
- If the **method does not return a value, its return type is void**.
- **methodName** - It is an identifier that is used to refer to the particular method in a program.
- **method body** - It includes the programming statements that are used to perform some tasks. The method body is **enclosed inside the curly braces { }**.

Declaring a Java Method

- For example,

```
int addNumbers() {
    // code
    return 1;
}
```

- In the above example, the name of the method is **addNumbers().** And, the **return type is int.**
- This is the simple syntax of declaring a method.

Declaring a Java Method

- However, the complete syntax of declaring a method is

```
modifier static returnType nameOfMethod (parameter1, parameter2, ...) {
  // method body
}
```

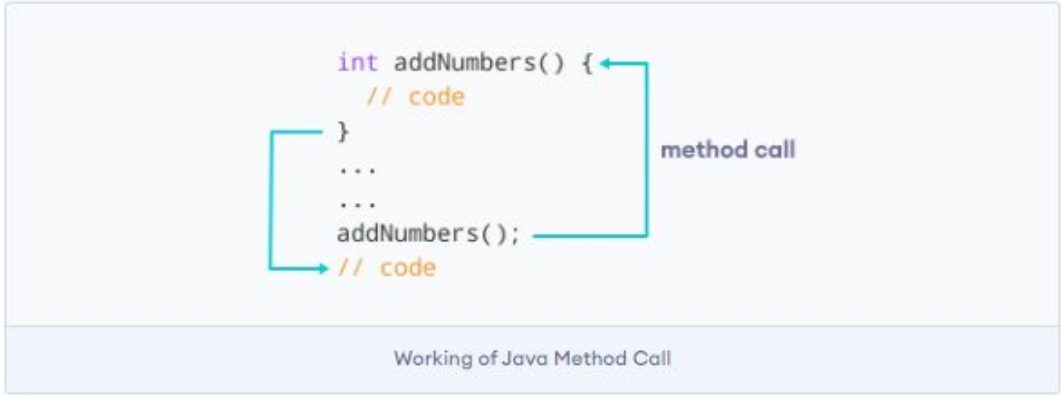
- **modifier** - It defines access types whether the method is public, private, and so on.
- **static** - If we use the static keyword, it can be accessed without creating objects.
- *For example, the sqrt() method of standard Math class is static. Hence, we can directly call Math.sqrt() without creating an instance of Math class.*
- **parameter1/parameter2** - These are values passed to a method. We can pass any number of arguments to a method.

Calling a Method in Java

- In the above example, we have declared a method named addNumbers().

  Now, to use the method, we need to call it.
- Here's is how we can call the addNumbers() method.

```
// calls the method
addNumbers();
```



Working of Java Method Call

Calling a Method in Java

- Example 1: Java Methods
- Output :

```
Sum is: 40
```

```
class Main {

  // create a method
  public int addNumbers(int a, int b) {
    int sum = a + b;
    // return value
    return sum;
  }

  public static void main(String[] args) {

    int num1 = 25;
    int num2 = 15;

    // create an object of Main
    Main obj = new Main();
    // calling method
    int result = obj.addNumbers(num1, num2);
    System.out.println("Sum is: " + result);
  }
}
```

Calling a Method in Java

- In the above example, we have created a method named **addNumbers().** The method takes **two parameters a and b**. Notice the line,

```
int result = obj.addNumbers(num1, num2);
```

- Here, we have **called the method by passing two arguments num1 and num2**. Since the method **is returning some value**, we have **stored the value in the result variable**.
- Note: *The method is not static. Hence, we are calling the method using the object of the class.*

Java Method Return Type

- A Java method **may or may not return a value** to the function call. We use the return statement to return any value. For example,

```
int addNumbers() {
...
return sum;
}
```

- Here, we are **returning the variable sum**. Since the **return type of the function is int**. The sum variable should be of **int** type. Otherwise, it will generate an error.

Java Method Return Type

- Example 2: Method Return Type
- Output :

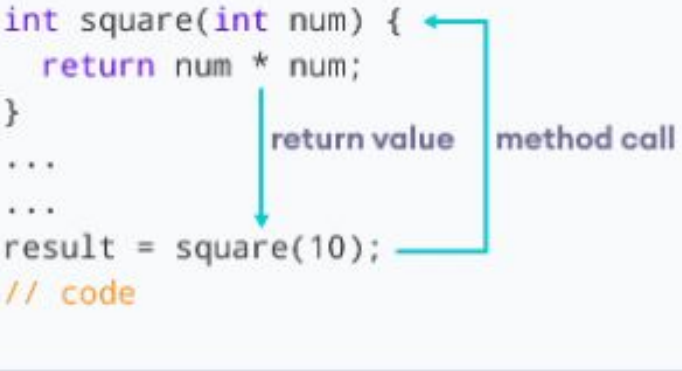
```
Squared value of 10 is: 100
```

- In the program, we have created a method named square().
- The method takes a number as its parameter and returns the square of the number.

```
class Main {
  // create a method
  public static int square(int num) {

    // return statement
    return num * num;
  }

  public static void main(String[] args) {
    int result;

    // call the method
    // store returned value to result
    result = square(10);

    System.out.println("Squared value of 10 is: " + result);
  }
}
```

Java Method Return Type

- Here, we have mentioned the return type of the method as int. Hence, the method should always return an integer value.



Representation of the Java method returning a value

Java Method Return Type

- If the **method does not return any value**, we use the **void** keyword as the return type of the method. For example,

```
public void square(int a) {
  int square = a * a;
  System.out.println("Square is: " +
a);
}
```

Method Parameters in Java

- A **method parameter is a value accepted by the method**. As mentioned earlier, a method can also have any number of parameters. For example,

```
// method with two parameters
int addNumbers(int a, int b) {
  // code
}

// method with no parameter
int addNumbers(){
  // code
}
```

Method Parameters in Java

- If a **method is created with parameters**, we need to **pass the corresponding values while calling the method**. For example,

```
// calling the method with two parameters
addNumbers(25, 15);

// calling the method with no parameters
addNumbers()
```

Method Parameters in Java

- Example 3: Method Parameters
- Output

```
Method without parameter
Method with a single parameter: 24
```

```
class Main {
  // method with no parameter
  public void display1() {
    System.out.println("Method without parameter");
  }

  // method with single parameter
  public void display2(int a) {
    System.out.println("Method with a single parameter: " + a);
  }

  public static void main(String[] args) {

    // create an object of Main
    Main obj = new Main();

    // calling method with no parameter
    obj.display1();

    // calling method with the single parameter
    obj.display2(24);
  }
}
```

Method Parameters in Java

- Here, the **parameter of the method is int**. Hence, **if we pass any other data type instead of int, the compiler will throw an error**. It is because Java is a strongly typed language.
- Note: **The argument 24 passed to the display2() method** during the method call is called the actual argument.
- **The parameter num accepted by the method definition is known as a formal argument**. We need to specify the type of formal arguments. And, the type of actual arguments and formal arguments should always match.

Standard Library Methods

- The **standard library methods are built-in methods in Java** that are readily available for use.
- These standard libraries come along with the Java Class Library (JCL) in a Java archive (*.jar) file with JVM and JRE.
- For example,
  - **print()** is a method of **java.io.PrintSteam.** The print("...") method prints the string inside quotation marks.
  - **sqrt()** is a method of **Math** class. It returns the square root of a number.

Standard Library Methods

- Example 4: Java Standard Library Method
- Output :

```
Square root of 4 is: 2.0
```

```
public class Main {
  public static void main(String[] args) {

    // using the sqrt() method
    System.out.print("Square root of 4 is: " + Math.sqrt(4));
  }
}
```

Advantages of using methods?

- The main advantage is **code reusability**.
- We can **write a method once, and use it multiple times**.
- We do not have to rewrite the entire code each time.
- Think of it as, **"write once, reuse multiple times".**

Advantages of using methods?

- Example 5: Java Method for Code Reusability
- Output

```
Square of 1 is: 1
Square of 2 is: 4
Square of 3 is: 9
Square of 4 is: 16
Square of 5 is: 25
```

```
public class Main {

  // method defined
  private static int getSquare(int x){
    return x * x;
  }

  public static void main(String[] args) {
    for (int i = 1; i <= 5; i++) {

      // method call
      int result = getSquare(i);
      System.out.println("Square of " + i + " is: " + result);
    }
  }
}
```

Advantages of using methods?

- In the above program, **we have created the method named getSquare()** to calculate the square of a number. Here, the **method is used to calculate the square of numbers less than 6**.
- Hence, **the same method is used again and again.**
- Methods make code more **readable and easier** to debug.
- Here, **the getSquare() method keeps the code to compute the square** in a block. **Hence, makes it more readable.**

TIA-Academy

ASSIGNMENT 02

TIA-Academy

Java Method Overloading

Java Method Overloading

- In Java, **two or more methods may have the same name if they differ in parameters** (different number of parameters, different types of parameters, or both). **These methods are called overloaded methods** and this feature is called method overloading. For example:

```
void func() { ... }
void func(int a) { ... }

float func(double a) { ... }
float func(int a, float b) { ... }
```

- Here, **the func() method is overloaded**. These **methods have the same name but accept different arguments.**

- Note: **The return types of the above methods are not the same**. It is because method overloading is not associated with return types. **Overloaded methods may have the same or different return types, but they must differ in parameters**.

Why method overloading?

- Suppose, you have to perform the addition of given numbers but there can be any number of arguments (let's say either 2 or 3 arguments for simplicity).
- In order to accomplish the task, you can create two methods **sum2num(int, int) and sum3num(int, int, int)** for two and three parameters respectively. However, other programmers, as well as you in the future may get confused as the behavior of both methods are the same but they differ by name.
- **The better way to accomplish this task is by overloading methods**. And, **depending upon the argument passed, one of the overloaded methods is called**. This helps to increase the readability of the program.

How to perform method overloading in Java?

- Overloading by changing the number of parameters
- Output:

```
Arguments: 1
Arguments: 1 and 4
```

```
class MethodOverloading {
    private static void display(int a){
        System.out.println("Arguments: " + a);
    }

    private static void display(int a, int b){
        System.out.println("Arguments: " + a + " and " + b);
    }

    public static void main(String[] args) {
        display(1);
        display(1, 4);
    }
}
```

How to perform method overloading in Java?

- Method Overloading by changing the data type of parameters
- Output:

```
Got Integer data.
Got String object.
```

- **Here, both overloaded methods accept one argument.**
- However, one accepts the argument of type **int** whereas other accepts **String** object.

```
class MethodOverloading {
    // this method accepts int
    private static void display(int a){
        System.out.println("Got Integer data.");
    }

    // this method  accepts String object
    private static void display(String a){
        System.out.println("Got String object.");
    }

    public static void main(String[] args) {
        display(1);
        display("Hello");
    }
}
```

How to perform method overloading in Java?

- Let's look at a real-world example:
- Output:

```
500
89.993
550.00
```

- Note: In Java, you can also **overload constructors in a similar way like methods.**.

```
class HelperService {
    private String formatNumber(int value) {
        return String.format("%d", value);
    }

    private String formatNumber(double value) {
        return String.format("%.3f", value);
    }

    private String formatNumber(String value) {
        return String.format("%.2f", Double.parseDouble(value));
    }

    public static void main(String[] args) {
        HelperService hs = new HelperService();
        System.out.println(hs.formatNumber(500));
        System.out.println(hs.formatNumber(89.9934));
        System.out.println(hs.formatNumber("550"));
    }
}
```

Important Points

- Two or more methods can have the same name inside the same class if they accept different arguments. This feature is known as **method overloading.**
- Method overloading is achieved by either:
  - changing the number of arguments.
  - or changing the data type of arguments.
- It is not method overloading if we only change the return type of methods. There must be differences in the number of parameters.

TIA-Academy

ASSIGNMENT 03
(HOME ASSIGNMENT)

TIA-Academy

Thank You