🔷 What is a function?

🔹 In Mathematics

In **mathematics**, a **function** is a rule that assigns exactly one output to each input, and the expression **$f(x) = x^2 + 2$** is a classic example of this. Here, for any value of **x** you plug in, the function **squares it and then adds 2**, always producing a **single, well-defined result**.

- For instance, if **$x = 3$**, then **$f(3) = 3^2 + 2 = 11$**
- If **$x = -1$**, then **$f(-1) = (-1)^2 + 2 = 3$** Since each input **x** gives only one output **$f(x)$**, this satisfies the definition of a function in mathematics.

🔹 In Java

In **Java**, a function (**called a method**) is a **block of code** that performs a specific task. You define it once and can use (**call**) it multiple times.

It is like a **coffee machine** in real life. When you press a button on the coffee machine and choose your **coffee type and sugar level**, the machine processes your input and gives you a **cup of coffee**. Similarly, in Java, you can write a method like **makeCoffee(String type, int sugarLevel)** that takes **inputs (parameters)**, performs a task (like making coffee), and **returns a result**. Instead of writing the coffee-making steps each time, you just **call the method** whenever you need it, possibly with different inputs — making your code **clean, reusable, and efficient**.

🔹 Reusability Example

For example, you can write a function to **check if a number is prime**, and even if you need to use it 20 times, you can simply **call it 20 times** instead of rewriting the logic each time. This improves **readability**, enhances **accessibility**, and makes the code much easier to **manage and maintain**.

🔹 Program Execution and Memory

The execution of a program begins with the **user**, but to actually run the program, **memory is required**. This memory is provided by the **operating system**, which acts as a **mediator between the user and the hardware**. The operating system plays a crucial role as a **resource manager**, allocating memory and managing system resources to ensure **smooth execution**.

🔹 Hardware Analogy

Now, can we watch a movie without hardware? Technically, we could write code to simulate the experience, but we'd end up spending more time developing features than enjoying the movie itself. This highlights the importance of **hardware and the operating system working together** to deliver a **seamless experience**.

🔹 Types of Memory

Memory used during program execution is generally divided into two types:

- **Stack memory** is responsible for **running the program and managing function calls**
- **Heap memory** is used for **dynamic memory allocation**, which we'll discuss in more detail later

🔹 Role of the Compiler

It's also important to clarify the role of the **compiler**. The compiler does **not execute the program**—it only checks for **syntax errors**, much like how the sentence "I are eat" is grammatically incorrect. The compiler ensures that the code follows the **correct structure and rules** of the programming language.

🔹 JVM (Java Virtual Machine)

The **JVM (Java Virtual Machine)** executes the program and provides the **environment for its execution**. This environment is like the **special atmosphere during a wedding in the house**—unlike regular days, it's filled with **lights, unlimited food, sweets, drinks, and celebration**.

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello Akarsh!");
    }
}
```

🔹 Uniqueness of main()

Imagine a unique person—he's wearing a **red T-shirt, orange jeans, green shoes**, and has **dusky hair**. If you go to a public place like **Gaur City Mall**, you'd likely find only one person dressed exactly like that. Similarly, in a Java program, the **main function is unique**—it exists only once. There cannot be **multiple main methods with the same signature** in a single class.

🔹 JVM and main()

The **JVM (Java Virtual Machine)** always starts executing a Java program from the **main function**. When the JVM loads a class to run, it looks for the **main method as the entry point**. There must be only one **main method with the correct signature** in a class—only a single instance of it can exist.

The JVM begins execution from this method and continues running the program from there. Also, the Java program is executed by the **JVM**, which runs the program from **outside the class**.

For the JVM to access the main method, it must be declared as **public**. If it's marked **private**, the JVM won't be able to see or access it, because private methods are only visible within the class. Declaring it **public** ensures that the JVM can **invoke it from outside the class**.

🔹 Return Types in Functions

Suppose I give you a task. You'll return once the task is complete. Now, there are two possible outcomes:

- You come back and say, "I've completed the task," but you don't bring anything with you.
- I ask you to buy some apples, and you return with those apples. In this case, you've brought something back.

This is exactly how **return types** work in functions.

- If a function doesn't return anything, its return type is **void**.
- But if it returns something—like **₹2000** from someone—the return type might be **int**.
- If it returns **24.5 liters of petrol**, the return type could be **float**.

So, the **return type** of a function depends on what it **brings back after completing its task**.

🔹 Why is main() void?

Now, why is the return type of the **main function in Java void**? Because Java doesn't expect anything in return from the **main method**. It's simply the **entry point of the program**, and once the execution is done, there's nothing that needs to be returned to the JVM.

🔹 Static Keyword (Preview)

We'll cover the **static keyword** in more detail when we learn about **OOP concepts**. But if I had to explain it in a single line:

The program is executed by the **JVM**, which exists **outside the program itself**. Since the JVM cannot create an object of the class just to start execution, it needs to **call the main method directly**. To call a method without creating an object, the method must be **static**. That way, it can be invoked using just the **class name**—no object required.

🔹 Types of Methods

Methods are basically of two types:

- **Parameterized**
- **Non-parameterized**

A parameterized method can have a return type of **void** or any of the **primitive data types** you've studied. The same applies to **non-parameterized methods** as well.

🔹 Method Placement in Java

If you come from another programming background, such as **C**, you might be used to defining functions above the **main method**. However, **Java is a flexible language**—you can define methods either **before or after the main method**. Their position in the code **doesn't affect the program's execution**.

```java
public class Main {

      public static void main(String[] args) {
            System.out.println("Hello Akarsh!");
            System.out.println("Bye Akarsh!");
      }

      public static void addition() {
            int a = 8;
            int b = 6;
            int c = a + b;
            System.out.println(c);
      }

}
```

🔹 Function Not Called

In the code example below, the output will be **'Hello'** and **'Bye'** because the **addition function is never called**.

- If a function is **not called**, it will **not execute** and will **not produce any output**.
- I could have created **multiple functions**, but unless they are **explicitly called**, they will **never be executed**.

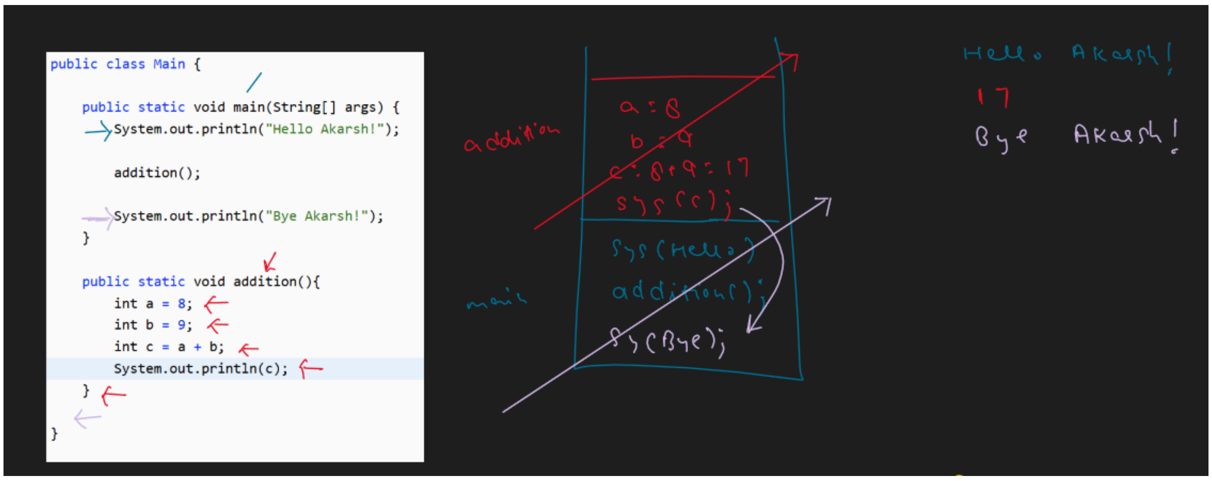
🔹 Call Stack

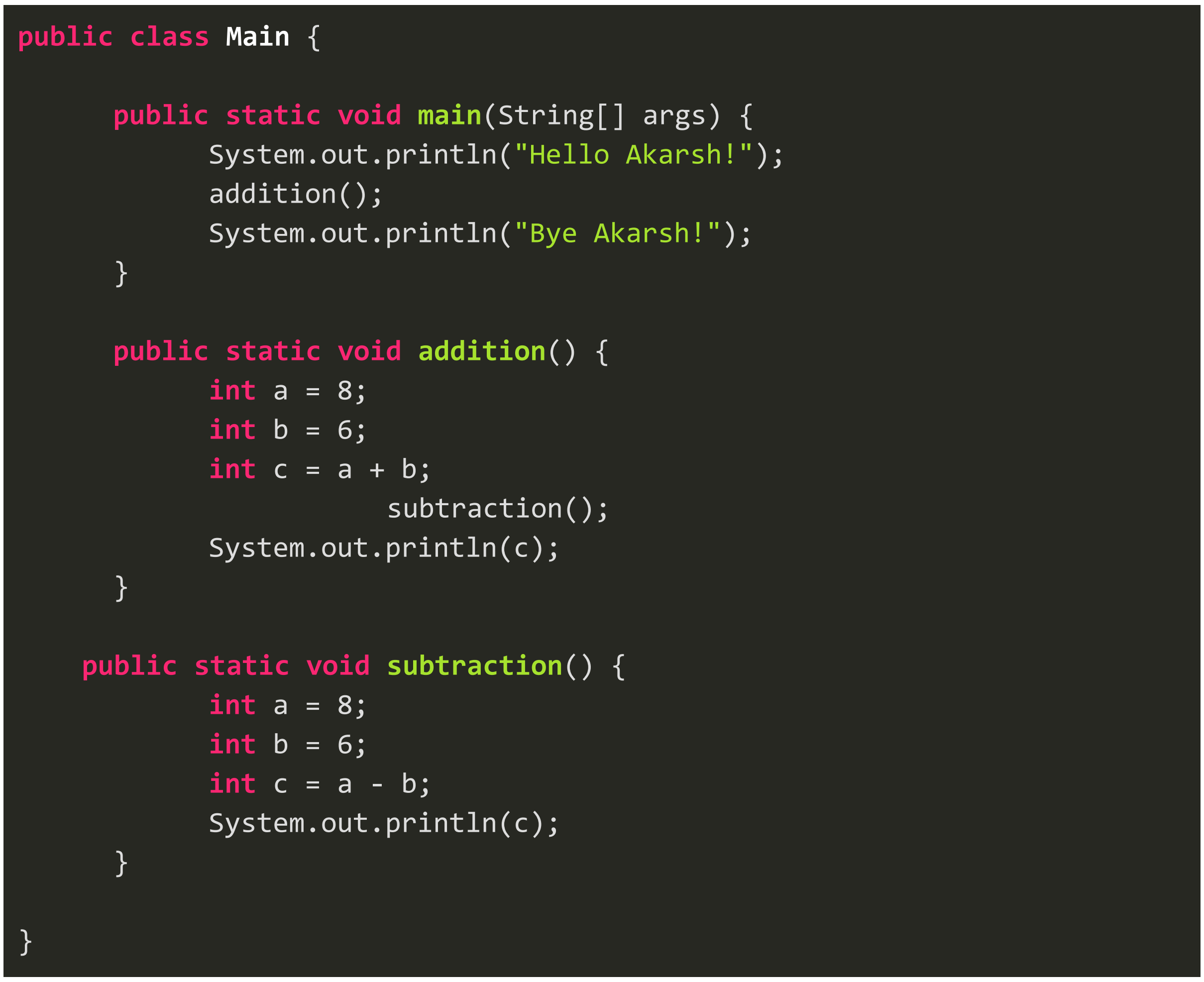
The **call stack** represents the **sequence of function calls** in a program. It shows:

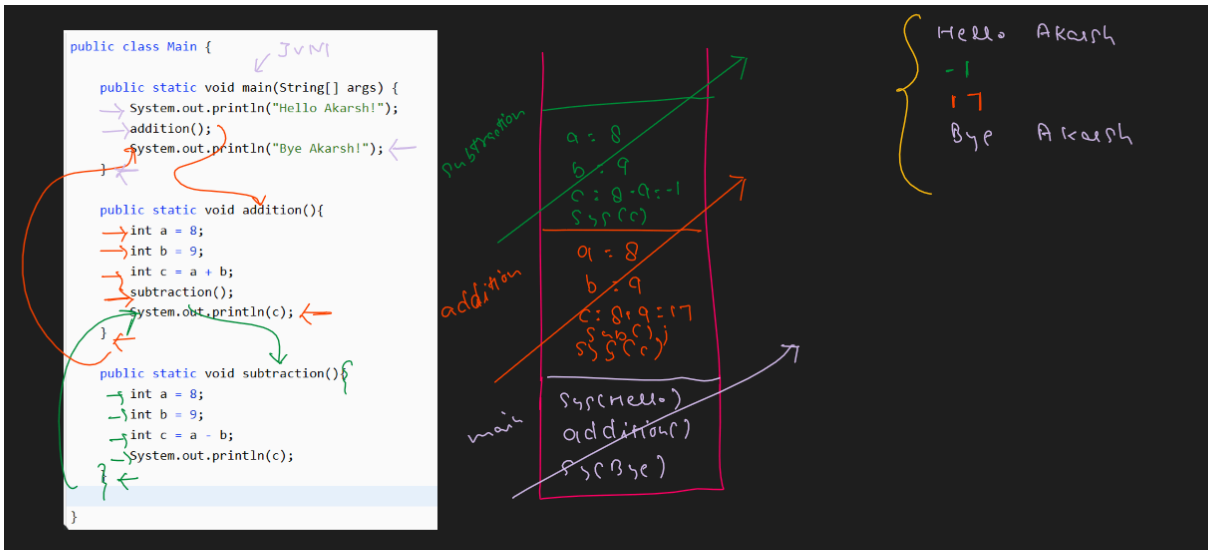
- Which function is **currently executing**
- How the program **arrived at that point**

Ex: Marathon race where the contestant passes the stick to each other, parcel delivery.

```java
public class Main {

      public static void main(String[] args) {
            System.out.println("Hello Akarsh!");
            addition();
            System.out.println("Bye Akarsh!");
      }

      public static void addition() {
            int a = 8;
            int b = 6;
            int c = a + b;
            System.out.println(c);
      }

}
```


```
public class Main {

    public static void main(String[] args) {
        System.out.println("Hello Akarsh!");

        addition();

        System.out.println("Bye Akarsh!");
    }

    public static void addition(){
        int a = 8;
        int b = 9;
        int c = a + b;
        System.out.println(c);
    }
}
```


```
public class Main {

      public static void main(String[] args) {
            System.out.println("Hello Akarsh!");
            addition();
            System.out.println("Bye Akarsh!");
      }

      public static void addition() {
            int a = 8;
            int b = 6;
            int c = a + b;
                        subtraction();
            System.out.println(c);
      }

    public static void subtraction() {
            int a = 8;
            int b = 6;
            int c = a - b;
            System.out.println(c);
      }

}
```



```java
public class Main {

    public static void main(String[] args) {
        System.out.println("Hello Akarsh!");
        addition();
        System.out.println("Bye Akarsh!");
    }

    public static void addition(){
        int a = 8;
        int b = 9;
        int c = a + b;
        subtraction();
        System.out.println(c);
    }

    public static void subtraction(){
        int a = 8;
        int b = 9;
        int c = a - b;
        System.out.println(c);
    }

}
```

🔹 Local Variables and Scope

Also, the same variables **a** and **b** are declared in the **addition function**, and then again in the **subtraction function**.

- You might wonder if they are the same—but they're **not**.
- They belong to **different scopes**.
- They are called **local variables**, defined in each respective function.

🔹 Analogy: Same Name, Different Houses

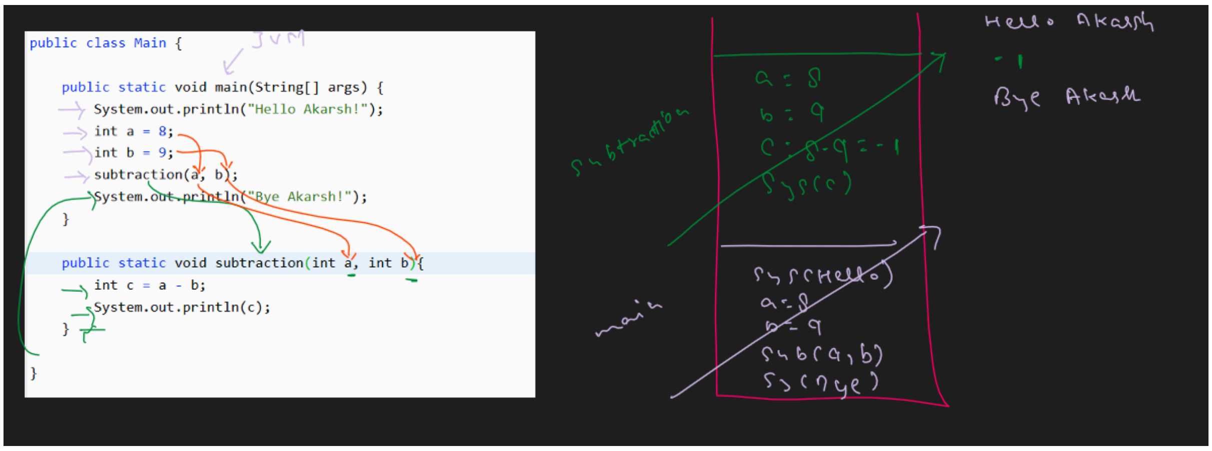
Think of it like this:

- Imagine a child named **Aman** who lives in **Sharma's house**, and another child named **Aman** who lives in **Verma's house**.
- Are they the same person? **No**, they're **different children with the same name**, living in **different houses**.

🎯 Similarly, variables with the **same name in different functions** are:

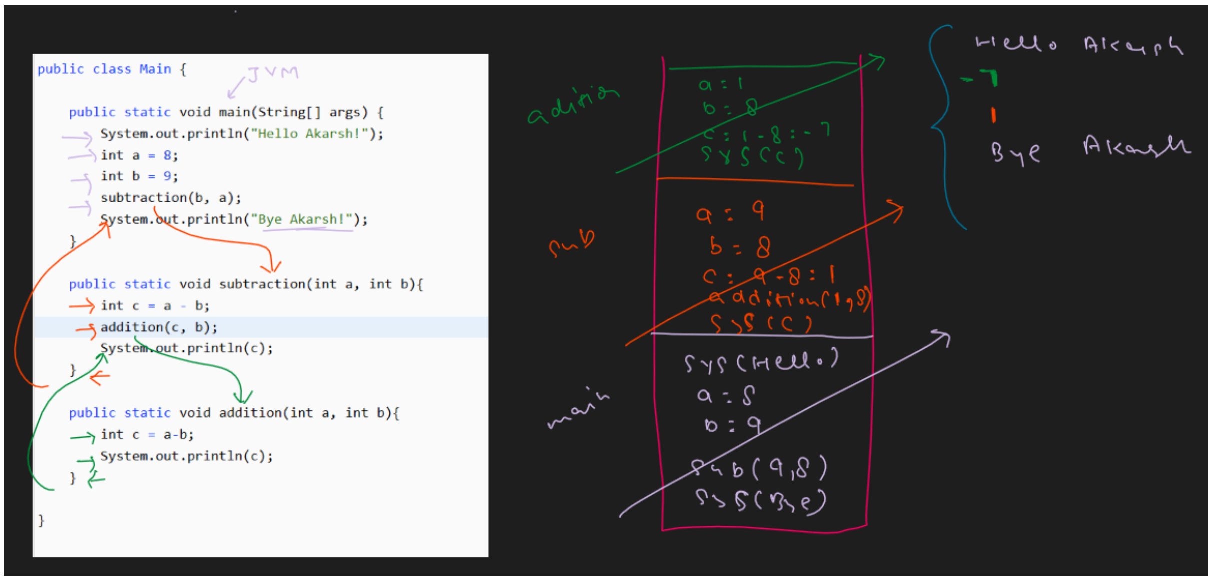
- **Separate and unrelated**
- They exist in **different scopes**

```java
public class Main {

      public static void main(String[] args) {
            System.out.println("Hello Akarsh!");
            int a = 9;
            int b = 8;
            sub(b, a);
            System.out.println("Bye Akarsh!");

      }

      public static void sub(int a, int b) {
            int c = a - b;
            System.out.println(c);
      }

}
```



```java
public class Main {

    public static void main(String[] args) {
        System.out.println("Hello Akarsh!");
        int a = 8;
        int b = 9;
        subtraction(a, b);
        System.out.println("Bye Akarsh!");
    }

    public static void subtraction(int a, int b){
        int c = a - b;
        System.out.println(c);
    }

}
```

```java
public class Main {

	public static void main(String[] args) {
		System.out.println("Hello Akarsh!");
		int a = 9;
		int b = 8;
		sub(b, a);
		System.out.println("Bye Akarsh!");

	}

	public static void sub(int a, int b) {
		int c = a - b;
		add(c, b); //comment and uncomment both scenarios
		System.out.println(c);
	}

	public static void add(int a, int b) {
		int c = a - b;
		System.out.println(c);
	}

}
```



```java
public class Main {

    public static void main(String[] args) {
        System.out.println("Hello Akarsh!");
        int a = 8;
        int b = 9;
        subtraction(b, a);
        System.out.println("Bye Akarsh!");
    }

    public static void subtraction(int a, int b){
        int c = a - b;
        addition(c, b);
        System.out.println(c);
    }

    public static void addition(int a, int b){
        int c = a-b;
        System.out.println(c);
    }

}
```

```java
public class Main {

        public static void main(String[] args) {
        System.out.println("Hello Akarsh!");
        int a = 8;
        int b = 9;
        // System.out.println(addition(a, b));
        int cc = addition(a, b);
        System.out.println("Printing sum of two numbers...");
        System.out.println(cc);
        System.out.println("Bye Akarsh!");
        }

        public static int addition(int a, int b){
        int c = a+b;
        return c;
        }

}
```

```java
public class Main {

        public static void main(String[] args) {
        System.out.println("Hello Akarsh!");
        int a = 8;
        int b = 9;
        int cc = addition(104, b);
        System.out.println(cc);
        System.out.println("Bye Akarsh!");
        }

        public static int addition(int a, int b){
        int c = a+b;
        return c;
        }

}
```

🔹 What Is a Global Variable?

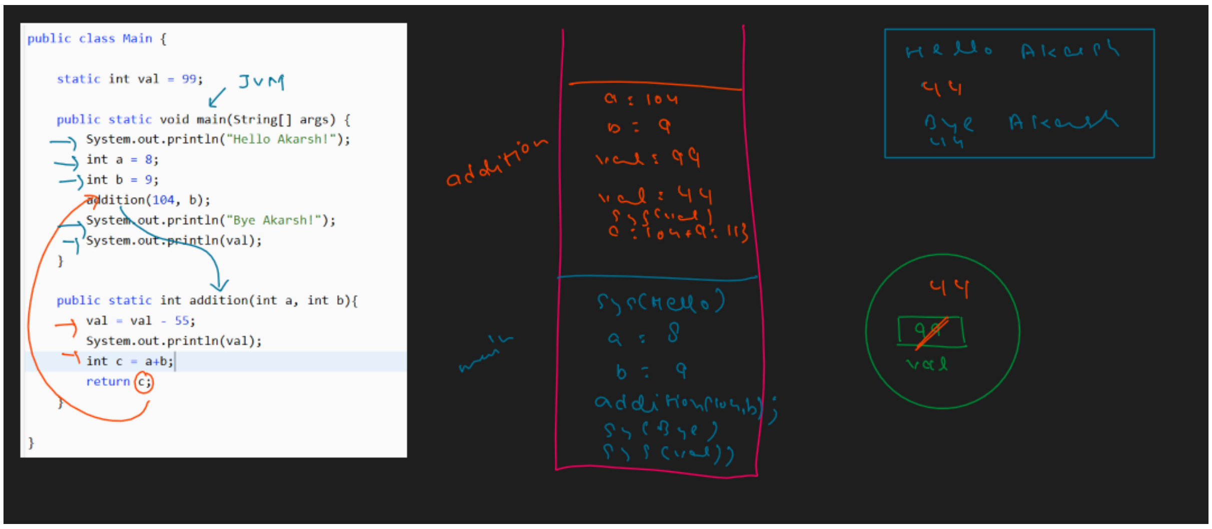
A **global variable** is like the **scoreboard at a gully cricket match**.

🔹 Analogy: The Shared Scoreboard

- Imagine a bunch of kids playing cricket in the park.
- There's one big **scoreboard written with chalk on a wall** — it shows the **total runs**.
- Everyone on both teams can **see it**, and anyone (usually the loudest kid) can **update it**.
- So, if **Bunty hits a six**, he runs over and **adds 6 to the total**.
- But sometimes, **Chintu tries to sneakily add extra runs** for his team when no one's watching!

🔹 Why It's Like a Global Variable

- The scoreboard is **visible and editable by everyone**.
- It's just like a **global variable — shared across the entire game (program)**.
- If one person (**function**) changes it:
  - The **whole team (other functions)** sees the new value.
  - ✅ Whether it's **right or wrong**, the change is reflected everywhere.



```java
public class Main {

    static int val = 99;

    public static void main(String[] args) {
        System.out.println("Hello Akarsh!");
        int a = 8;
        int b = 9;
        addition(104, b);
        System.out.println("Bye Akarsh!");
        System.out.println(val);
    }

    public static int addition(int a, int b){
        val = val - 55;
        System.out.println(val);
        int c = a+b;
        return c;
    }

}
```

🔹 Local vs Global Variable Priority

Imagine there’s a popular **chaiwala named Ramesh** outside an office — everyone in the building knows him (he's the "**global**" Ramesh).

🔹 The Confusion

- One day, during a **team meeting**, the manager says, “**Ramesh, get us tea.**”
- But there’s also an **intern named Ramesh** sitting right there in the meeting (the "**local**" Ramesh).
- Without thinking, the **intern jumps up** and says, “**On it, sir!**” — and rushes to get the tea.
- Meanwhile, the **actual chaiwala** is still outside doing his job.

🔹 Programming Insight

This shows how, in programming:

- When a **local and global variable** have the **same name**,
- The **local one gets priority** inside that specific **function or block** —
- Even if the **global one is more widely recognized**.

```java
public class Main {

    static int val = 99;

    public static void main(String[] args) {
        System.out.println("Hello Akarsh!");
        int a = 8;
        int b = 9;
        System.out.println(val);
        addition();
        System.out.println("Bye Akarsh!");
        System.out.println(val);
    }

    public static void addition(){
        int val = 8;
        val = val - 55;

        System.out.println("Global: " + Main.val);
        System.out.println(val);
    }
}
```

🔹 Accessing Global Variables Explicitly

ClassName.variableName → always the **global variable** will be accessed.

🔹 Practice Problems

Is Armstrong Number

```java
import java.util.Scanner;

public class Main {

      public static void main(String[] args) {
            Scanner sc = new Scanner(System.in);
            int n = sc.nextInt();
            System.out.println(isArmstrong(n));
      }

      public static boolean isArmstrong(int n) {
            int cod = countOfDigit(n);
            int sum = 0;
            int tempN = n;

            while (n > 0) {
                  int rem = n % 10;
                  sum = (int) (sum + Math.pow(rem, cod));
                  n = n / 10;
            }

            if (sum == tempN) {
                  return true;
            } else {
                  return false;
            }
      }

      public static int countOfDigit(int n) {
            int count = 0;
            while (n > 0) {
                  count++;
                  n = n / 10;
            }
            return count;
```

```
        }

}
```