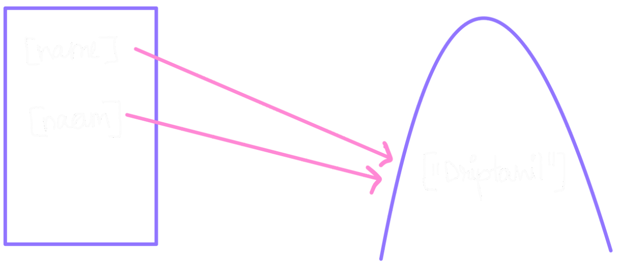
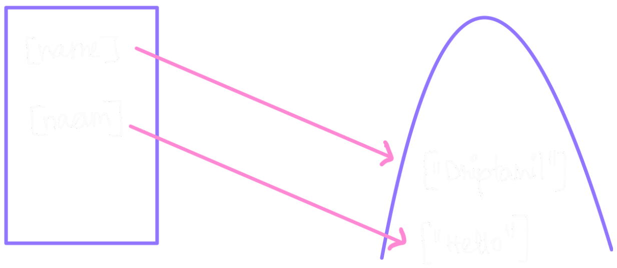
Functions

```java
public class Greeting {
    public static void ain(String[] args) {
        String name = "Driptanil";
        String personalised = myGreet(name);
        System.out.println(personalised);
    }

    static String myGreet(String naam) {
        String message = "Hello " + naam;
        return message;
    }
}
/* Output : Hello Driptanil */
```

- `String` is the return type of `myGreet` function
- `String name` is the input type and local variable name and is called parameters of the function.

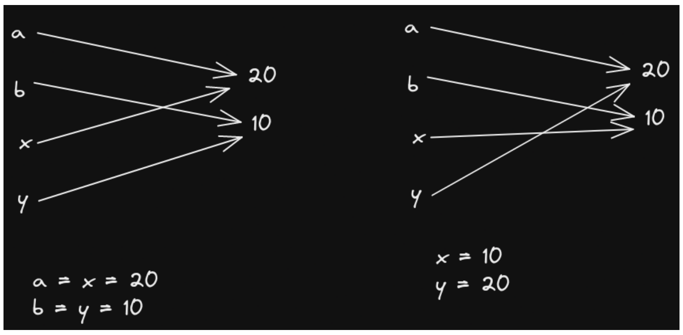
```java
public class Greeting {
    public static void main(String[] args) {
                                String name = "Driptanil";
                                func(name);
                                System.out.println(name);
    }

    static void func(String naam) {
        String naam = "Hello";
    }
}

/* Output : Driptanil */
```





- Strings cannot be changed, because Strings are immutable. So, new Strings are created in heap memory
- `naam` variable can be only called in the scope of `func` function, calling `naam` in others function will give an error. This is called Scoping.
- In java, `a = b` a is passed by the value of b [not reference]

```java
public class Swap {
    public static void main(String[] args) {
        int a = 20;
        int b = 10;
        swap(a,b);
        System.out.println(a+" "+b);
    }

    static void swap(int x, int y) {
        int temp = x;
        x = y;
        y = temp;
    }
}

/* Output : 20 10 */
```



Although `x` and `y` are modified, `a` and `b` remain unchanged, because maybe new objects are made in heap memory.

This is applicable for Primitive data types.

```
import java.util.Arrays;

public class Array {
    public static void main(String[] args) {
        int[] arr = {1, 3, 5, 7, 9};
        func(arr);
        System.out.println(Arrays.toString(arr));
    }

    static void func(int[] array) {
        array[0] = 11;
    }
}

/* Output : [11, 3, 5, 7, 9] */
```

This does not work for non-primitive data-types.

But this is not the same for arrays. Although `array` is changed, `arr` also gets changed, but in the case of arrays new objects are not created in heap memory.

Scope

```
public class Scope {
  public static void main(String[] args) {
        int a = 10;
        int b = 20;
        String name = "Kunal";
        {
//                int a = 78; // already initialised outside the block in the same method, hence
you cannot initialise again
                a = 100; // reassign the origin ref variable to some other value
            System.out.println(a);
                int c = 99;
                name = "Rahul";
            System.out.println(name);
        }
//        System.out.println(c); // cannot use outside the block
        int c = 900;
        System.out.println(a);
        System.out.println(name);

        // scoping in for loops
        for (int i = 0; i < 4; i++) {
//            System.out.println(i);
            int num = 90;
            a = 10000;
        }
        System.out.println();
  }

  static void random(int marks) {
        int num = 67;
        System.out.println(num);
        System.out.println(marks);
  }
}

/* Output :
                                100
                                Rahul
                                100
                                Rahul */
```

- values declared in this block, will remain in block

Shadowing

```
public class Shadowing {
    static int x = 90; // this will be shadowed at line 8
    public static void main(String[] args) {
        System.out.println(x); // 90
        int x = 40;
        System.out.println(x); // 40
        fun();
    }

    static void fun() {
        System.out.println(x);
    }
}
```

Here `x` is declared in class block, this allows calling `x` variable in any function in class `Shadowing`.

If `x` is again declared in any function in `Shadowing` class, and calling of `x` variable in the same function, would give value initialised to `x` variable in the function. i.e. value of `x` initialised in `Shadowing` class would get shadowed. This is called Shadowing.

But when `x` variable is called in a function that hasn't declared or initialised any value to `x` variable, would give the value initialised to `x` variable in `Shadowing` class.

Variable Arguments

```java
package com.inclass;

import java.util.Arrays;

public class VarArgs {
    public static void main(String[] args) {
        args(1, 2, 3, 4, 5, 6, 7, 8);
    }

    static void args(int ...v) {
        System.out.println(Arrays.toString(v));
    }
}

/* Output : [1, 2, 3, 4, 5, 6, 7, 8] */
```

Variable Arguments are stored in heap as an array of primitive data types.

Variable Argument has to be used after all parameters in a function.

Function Overloading

```java
package com.inclass;

public class FunctionOverloading {
    public static void main(String[] args) {
        String name ="Driptanil";
        print();
        print(name);
    }

    static void print() {
        System.out.print("Hello ! ");
    }

    static void print(String name) {
        System.out.println(name);
    }
}

/* Output : Hello ! Driptanil */
```

Use of functions with same name but different parameters is called Function Overloading.

Ambiguity

```java
package com.inclass;

import java.util.Arrays;

public class Ambigutity {
    public static void main(String[] args) {
        func(); // ERROR !
    }

    static void func(int ...v) {
        System.out.println(Arrays.toString(v));
    }

    static void func(String ...v) {
        System.out.println(Arrays.toString(v));
    }
}
```

When more than one variable arguments are present in function overloading, then calling a function with no parameters would give a error. This is called Ambiguity.