***Functions in Java***

In Java, functions (also known as methods) are blocks of code that perform a specific task, are reusable, and can be called from other parts of a program. They are defined within a class and help in organizing code, making it more readable, maintainable, and reusable.

**Why We Use Functions**

* Breaks down complex problems into smaller, more manageable parts.
* Once a function is written, it can be reused multiple times in different parts of the program.
* Makes code easier to read, understand, and maintain.
* Hides the implementation details and only exposes the functionality, making it easier to work with.

**Syntax**

accessModifier returnType functionName(parameters) {

// function body

// statements

return value; // if returnType is not void

}

**Example:**

public class FunctionExample {

// Function to add two numbers

public static int add(int num1, int num2) {

int sum = num1 + num2;

return sum;

}

public static void main(String[] args) {

int a = 5;

int b = 10;

// Calling the add function

int result = add(a, b);

System.out.println("The sum is: " + result); // Output: The sum is: 15

}

}

***Types of Functions:***

1. Based on Return type: Void method, non-Void Method
2. Based on Parameter: Parameter less Method, Parameterized Method
3. Based on Access Specifier: Public, Private, Protected

Difference between passed by value and passed by reference.

|  |  |  |
| --- | --- | --- |
| Aspect | Void Method | Non-Void Method |
| Definition | A method that does not return any value. | A method that returns a value of a specified type. |
| Syntax | public void methodName() { } | public returnType methodName() { return value; } |
| Return Statement | No return statement needed. | Requires a return statement that matches the return type. |
| Usage | Typically used for performing actions or operations, such as printing or modifying object states. | Used when a result needs to be computed and returned, such as a calculation or data retrieval. |
| Example | public void printMessage() { System.out.println("Hello World"); } | public int add(int a, int b) { return a + b; } |
| Calling Method | object.printMessage(); | int result = object.add(5, 3); |
| Impact on Caller | Does not provide any data back to the caller. | Provides data back to the caller which can be used for further processing. |
| Common Use Cases | Initializing values, updating UI elements, logging information. | Performing calculations, retrieving data from a database, returning the result of a process. |

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

|  |  |  |
| --- | --- | --- |
| Aspect | Parameter-less Method | Parameterized Method |
| Definition | A method that does not take any parameters. | A method that takes one or more parameters. |
| Syntax | public void methodName() { } | public void methodName(int param1, String param2) { } |
| Purpose | Used when the method does not need any input to perform its task. | Used when the method requires input values to perform its task. |
| Usage | Often used for actions that are independent of input data, like printing a static message. | Used for actions that depend on input data, like calculations or operations based on the provided arguments. |
| Example | public void printMessage() { System.out.println("Hello World"); } | public void printMessage(String message) { System.out.println(message); } |
| Calling Method | object.printMessage(); | object.printMessage("Hello World"); |
| Flexibility | Less flexible as the operation is fixed. | More flexible as the operation can vary based on input parameters. |
| Common Use Cases | Displaying static content, initializing certain states. | Performing operations that depend on user input, processing data, or calculations. |

|  |  |  |  |
| --- | --- | --- | --- |
| Aspect | Public Method | Private Method | Protected Method |
| Definition | Methods that can be accessed from any other class. | Methods that can only be accessed within the same class. | Methods that can be accessed within the same package and subclasses. |
| Syntax | ‘public void methodName() { }’ | ‘private void methodName() { }’ | ‘protected void methodName() { }’ |
| Access Level | Least restrictive; accessible from any other class. | Most restrictive; accessible only within the class it's defined. | Moderately restrictive; accessible within the package and by subclasses. |
| Inheritance | Can be inherited and overridden by subclasses. | Cannot be inherited or overridden by subclasses. | Can be inherited and overridden by subclasses, even if they are in different packages. |
| Usage | Used when the method needs to be accessible from any other class, like utility methods. | Used for methods that should be hidden from other classes and are only used internally. | Used when the method needs to be accessible to subclasses and classes within the same package. |
| Example | public void display() { } | private void calculate() { } | protected void initialize() { } |
| Visibility | Visible everywhere. | Visible only within the class. | Visible within the same package and in subclasses. |
| Encapsulation | Provides less encapsulation. | Provides maximum encapsulation. | Provides moderate encapsulation. |

|  |  |  |
| --- | --- | --- |
| Aspect | Passed by Value | Passed by Reference |
| Definition | A copy of the actual value is passed to the function. Changes made to the parameter inside the function do not affect the original variable. | A reference to the actual variable is passed to the function. Changes made to the parameter affect the original variable. |
| Java Support | Java supports passing by value only. Objects are passed by value of the reference, not by reference. | Java does not support passing by reference directly. |
| Behavior with Primitives | For primitive data types (int, float, etc.), the actual value is passed. Changes to the parameter do not affect the original value. | N/A |
| Behavior with Objects | For objects, the reference (address) is passed by value. Changes to the object's fields affect the original object, but reassigning the object reference does not affect the original reference. | N/A |
| Example (Primitive Type) | java public void modifyValue(int x) { x = 10; } int a = 5; modifyValue(a); // a is still 5 | N/A |
| Example (Object Type) | java public void modifyObject(MyObject obj) { obj.value = 10; } MyObject myObj = new MyObject(); myObj.value = 5; modifyObject(myObj); // myObj.value is now 10 | N/A |
| Effect on Original Data | The original data is not affected when a primitive is passed. | When an object is passed, the fields of the object can be changed, but the reference itself remains unchanged. |
| Terminology Clarification | The term "pass by reference" is often used informally to describe the behavior of passing object references by value. | Strictly speaking, pass by reference means passing the actual reference itself, which Java does not do. |