**Methods:**

A **method** is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

Create a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses **()**. Java provides some pre-defined methods, such as System.out.println(), but you can also create your own methods to perform certain actions:

public class Main {

static void myMethod() {

// code to be executed

}

}

## Call a Method

To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon**;**

public class Main {

static void myMethod() {

System.out.println("I just got executed!");

}

public static void main(String[] args) {

myMethod();

}

}

// Outputs "I just got executed!"

A method can also be called multiple times:

public class Main {

static void myMethod() {

System.out.println("I just got executed!");

}

public static void main(String[] args) {

myMethod();

myMethod();

myMethod();

}

}

// I just got executed!

// I just got executed!

// I just got executed!

# Java Method Parameters

public class Main {

static void myMethod(String fname) {

System.out.println(fname + " Refsnes");

}

public static void main(String[] args) {

myMethod("Liam");

myMethod("Jenny");

myMethod("Anja");

}

}

// Liam Refsnes

// Jenny Refsnes

// Anja Refsnes

## Multiple Parameters

public class Main {

static void myMethod(String fname, int age) {

System.out.println(fname + " is " + age);

}

public static void main(String[] args) {

myMethod("Liam", 5);

myMethod("Jenny", 8);

myMethod("Anja", 31);

}

}

// Liam is 5

// Jenny is 8

// Anja is 31

## Return Values

The void keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as int, char, etc.) instead of void, and use the return keyword inside the method:

public class Main {

static **int** myMethod(int x) {

**return** 5 + x;

}

public static void main(String[] args) {

System.out.println(myMethod(3));

}

}

// Outputs 8 (5 + 3)

public class Main {

static int myMethod(int x, int y) {

return x + y;

}

public static void main(String[] args) {

int z = myMethod(5, 3);

System.out.println(z);

}

}

// Outputs 8 (5 + 3)

## Method Overloading

With**method overloading**, multiple methods can have the same name with different parameters:

Consider the following example, which has two methods that add numbers of different type:

static int plusMethodInt(int x, int y) {

return x + y;

}

static double plusMethodDouble(double x, double y) {

return x + y;

}

public static void main(String[] args) {

int myNum1 = plusMethodInt(8, 5);

double myNum2 = plusMethodDouble(4.3, 6.26);

System.out.println("int: " + myNum1);

System.out.println("double: " + myNum2);

}

Instead of defining two methods that should do the same thing, it is better to overload one.

In the example below, we overload the plusMethod method to work for both int and double:

static int plusMethod(int x, int y) {

return x + y;

}

static double plusMethod(double x, double y) {

return x + y;

}

public static void main(String[] args) {

int myNum1 = plusMethod(8, 5);

double myNum2 = plusMethod(4.3, 6.26);

System.out.println("int: " + myNum1);

System.out.println("double: " + myNum2);

}

Methods are blocks of code within a class that perform a specific task. They define the behavior of objects created from the class.

**// Adding a method to the Car class**

public void accelerate() {

System.out.println("The " + year + " " + brand + " is accelerating.");

}

**// Calling the new method on the Car object**

myCar.accelerate();

**Packages:**

A package is a way to organize related classes and interfaces into a single unit. It helps in avoiding naming conflicts and provides a hierarchical structure to the code.

**// Example of a package statement at the beginning of a file**

package com.example.myapp;

**// Importing a class from another package**

import java.util.ArrayList;

1. **ACCESS MODIFIER:**

In Java, access modifiers control the visibility of classes, fields, methods, and constructors in different parts of your code.

There are four main access modifiers: **public, private, protected, and the default (package-private).** Here's a simple explanation of the differences between private, public, and the default (package-private):

**private:**

Visibility: Limited to the class in which it is declared.

Access from Outside: Members (fields or methods) marked as private cannot be accessed directly from outside the class.

**Example:**

public class MyClass {

private int privateField;

private void privateMethod() {

// Code here

}

}

**public:**

Visibility: Accessible from any other class.

Access from Outside: Members marked as public can be accessed from any other class, including those in different packages.

**Example:**

public class MyClass {

public int publicField;

public void publicMethod() {

// Code here

}

}

**Default (Package-Private):**

Visibility: Accessible only within the same package.

Access from Outside: Members without any access modifier (default) can be accessed by classes within the same package but not by classes outside that package.

**Example:**

class MyClass {

int defaultField;

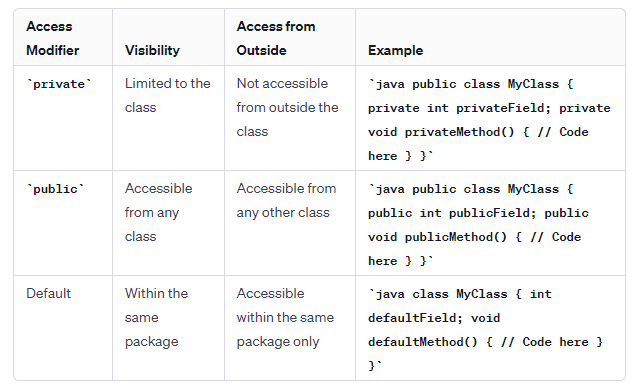
void defaultMethod() {

// Code here

}

}

It's worth noting that there is also an protected access modifier, which allows access within the same package and by subclasses even if they are in different packages. Using the appropriate access modifier is essential for encapsulation and managing the visibility of your code based on your design requirements.



In Java**, there are four access modifiers: public, protected, default (package-private), and private.** These modifiers control the visibility and accessibility of classes, variables, and methods. Here are examples for each access modifier:

**1. public:**

public class PublicExample {

public int publicVariable = 10;

public void publicMethod() {

System.out.println("This is a public method.");

}

}

In this example, the class PublicExample has a public variable and a public method. They can be accessed from anywhere in the program.

**2. protected:**

public class Parent {

protected int protectedVariable = 20;

protected void protectedMethod() {

System.out.println("This is a protected method.");

}

}

public class Child extends Parent {

public void accessProtected() {

System.out.println("Accessing protected variable: " + protectedVariable);

protectedMethod();

}

}

The protected access modifier allows access within the same package and by subclasses. In this example, the Child class inherits from Parent and can access its protected members.

**3. Default (Package-Private):**

class DefaultExample {

int defaultVariable = 30;

void defaultMethod() {

System.out.println("This is a default (package-private) method.");

}

}

If no access modifier is specified (package-private), it is visible only within the same package. In this example, the class DefaultExample is not declared with an access modifier, making it accessible only within the same package.

**4. private:**

public class PrivateExample {

private int privateVariable = 40;

private void privateMethod() {

System.out.println("This is a private method.");

}

public void accessPrivate() {

System.out.println("Accessing private variable: " + privateVariable);

privateMethod();

}

}

Members with the private modifier are accessible only within the same class. In the PrivateExample class, the accessPrivate method is used to access the private members.

These examples demonstrate the use of different access modifiers in Java. Remember that the choice of access modifier depends on the desired level of encapsulation and the requirements of your program.

**1. Class:**

A class is a blueprint for creating objects. It defines the properties and behaviors that objects of the class will have.

java

**// Example of a simple class**

public class Car {

**// Fields (variables)**

String brand;

int year;

**// Constructor**

public Car(String brand, int year) {

this.brand = brand;

this.year = year;

}

**// Method**

public void start() {

System.out.println("The " + year + " " + brand + " is starting.");

}

}

**2. Object:**

An object is an instance of a class. It represents a real-world entity based on the blueprint defined by the class.

**// Creating an object of the Car class**

Car myCar = new Car("Toyota", 2022);

**// Accessing object properties and calling methods**

System.out.println("My car is a " + myCar.year + " " + myCar.brand);

myCar.start();

1. **Simple example to illustrate the concept of return types in Java.**

In Java, a method can have a return type, indicating the type of value the method will return. Here's an example:

public class Calculator {

// Method to add two numbers and return the result

public static int add(int a, int b) {

return a + b;

}

// Method to multiply two numbers and return the result

public static double multiply(double x, double y) {

return x \* y;

}

public static void main(String[] args) {

// Using the add method and storing the result in a variable

int sum = add(5, 7);

System.out.println("Sum: " + sum);

// Using the multiply method and directly printing the result

System.out.println("Product: " + multiply(3.5, 2.0));

}

}

**In this example:**

* The Calculator class has two methods: **add and multiply**.
* The add method takes two integers as parameters, adds them, and returns the result as an int.
* The multiply method takes two doubles as parameters, multiplies them, and returns the result as a double.
* In the main method, we call the add and multiply methods and use the returned values.
* This example demonstrates the concept of return types. **The add method returns an int, and the multiply method returns a double. When you call these methods, you can capture and use the returned values in your program.**

1. **In Java, a constructor is a special method used to initialize the state of an object when it is created.**

public class Person {

// Fields (variables)

String name;

int age;

// Constructor

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Method to display information about the person

public void displayInfo() {

System.out.println("Name: " + name);

System.out.println("Age: " + age);

}

public static void main(String[] args) {

// Creating an object of the Person class using the constructor

Person person1 = new Person("John Doe", 25);

// Calling the displayInfo method to show information about the person

person1.displayInfo();

}

}

**Explanation:**

* The Person class has two fields (name and age) to represent a person's information.
* The class has a constructor public Person(String name, int age) that initializes the name and age fields when a Person object is created.
* The displayInfo method is used to print the information about the person to the console.
* In the main method, an object of the Person class is created using the constructor with the values "John Doe" and 25.
* The displayInfo method is called on the person1 object to display the information.
* When you run this program, it will create a Person object, initialize its fields using the constructor, and then display the information about that person.

**The output should be: Name: John Doe Age: 25**

**In Java, access modifiers are keywords that define the visibility or accessibility of classes, methods, and fields. There are four types of access modifiers:**

**Public:**

Public members are accessible from any class.

**Example:**

public class PublicExample {

public int publicVar;

public void publicMethod() {

System.out.println("This is a public method.");

}

}

**Private:**

Private members are accessible only within the same class.

**Example:**

public class PrivateExample {

private int privateVar;

private void privateMethod() {

System.out.println("This is a private method.");

}

}

**Protected:**

Protected members are accessible within the same class, subclass, and package.

**Example:**

public class ProtectedExample {

protected int protectedVar;

protected void protectedMethod() {

System.out.println("This is a protected method.");

}

}

**Default (Package-Private):**

If no access modifier is specified, it is considered package-private.

Default members are accessible within the same package.

**Example:**

class DefaultExample {

int defaultVar;

void defaultMethod() {

System.out.println("This is a default method.");

}

}

**Now, let's create a separate class to demonstrate the use of these access modifiers:**

public class AccessModifierDemo {

public static void main(String[] args) {

// Public example

PublicExample publicObj = new PublicExample();

publicObj.publicVar = 10;

publicObj.publicMethod();

// Private example (Cannot be accessed outside the class)

// PrivateExample privateObj = new PrivateExample();

// privateObj.privateVar; // This would result in a compilation error

// privateObj.privateMethod(); // This would result in a compilation error

// Protected example (Accessible in the same package)

ProtectedExample protectedObj = new ProtectedExample();

protectedObj.protectedVar = 20;

protectedObj.protectedMethod();

// Default example (Accessible in the same package)

DefaultExample defaultObj = new DefaultExample();

defaultObj.defaultVar = 30;

defaultObj.defaultMethod();

}

}

1. **Concept of the static keyword**

In Java, the static keyword is used to create class members (fields or methods) that belong to the class itself rather than to instances of the class. Here's a simple example to illustrate the concept of the static keyword:

**Example 1: Static Variable**

public class StaticExample {

// Static variable

static int staticVariable = 10;

public static void main(String[] args) {

// Accessing static variable directly using class name

System.out.println("Static Variable: " + StaticExample.staticVariable);

// Creating objects

StaticExample obj1 = new StaticExample();

StaticExample obj2 = new StaticExample();

// Modifying the static variable using an object (not recommended)

obj1.staticVariable = 20;

// Accessing static variable through different objects

System.out.println("Static Variable (obj1): " + obj1.staticVariable);

System.out.println("Static Variable (obj2): " + obj2.staticVariable);

}

}

In this example, staticVariable is a static variable. Even though we create two objects (obj1 and obj2), the static variable is shared among all instances of the class.

**Example 2: Static Method**

public class StaticExample {

// Static variable

static int staticVariable = 10;

// Static method

static void staticMethod() {

System.out.println("This is a static method.");

}

public static void main(String[] args) {

// Accessing static method directly using class name

StaticExample.staticMethod();

// Creating an object to access static method (not recommended)

StaticExample obj = new StaticExample();

obj.staticMethod(); // This works, but it's not a good practice

// Accessing static variable through different objects

System.out.println("Static Variable: " + obj.staticVariable);

}

}

In this example, staticMethod is a static method. It can be called using the class name without creating an instance of the class. However, calling static methods through objects is allowed (though not recommended).

Remember, static members are shared among all instances of the class, and they are associated with the class itself rather than with individual objects.