# What is Java used for?



Creating mobile and web apps
Creating enterprise software
Creating Internet of Things (IoT) devices
Creating gaming applications
Creating big data applications
Creating cloud-based applications

# Example



```
Int age = 18;

if (age >= 18) {
    System.out.println(“You are eligible to vote.”);
} else {
    System.out.println(“You are not eligible to vote yet.”);
}
```

# EqualsIgnore case



2. **equalsIgnoreCase() method:**

Same as equals(), but ignores the case of the strings.

Returns true if the strings are equal (ignoring case), false otherwise.

Example

String str1 = “Hello”;

String str2 = “hElLo”;

System.out.println(str1.equalsIgnoreCase(str2)); // Output: true

# 5. == operator:



**5. == operator**:

Checks if two string references point to the same object in memory.

Generally, not recommended for string comparison, as it can lead to unexpected results when comparing string literals or strings created using different methods.

**Example**

```
String str1 = “Hello”;
String str2 = “Hello”;
String str3 = new String(“Hello”);

System.out.println(str1 == str2); // Output: true (both refer to the same literal)
System.out.println(str1 == str3); // Output: false (different objects in memory)
```

# Example



```
Int x = 10;
String result = (x > 5) ? “x is greater than 5” : “x is less than or equal to 5”;
 System.out.println(result); // Output: x is greater than 5
```

# Example



```
Function greet(name) {
  console.log(“Hello, “ + name + “!”);
}

greet(“Alice”); // Output: Hello, Alice!
Greet(“Bob”);   // Output: Hello, Bob!
```

**In this example:**

The function is named greet.

It takes one parameter, name.

The code inside the function logs a greeting message using the value passed to the name parameter.

# Methods overloading in java



Method overloading in Java allows you to define multiple methods in the same class with the same name, but with different parameters. The compiler determines which method to call based on the number, types, and order of the arguments passed during the method invocation.

# Example



```
Public class Calculator {

    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }

    public int add(int a, int b, int c) {
        return a + b + c;
    }
}

public class Main {

    public static void main(String[] args) {
        Calculator calc = new Calculator();

        System.out.println (calc.add(2, 3));     // Output: 5
        System.out.println (calc.add(2.5, 3.5));  // Output: 6.0
        System.out.println (calc.add(1, 2, 3));   // Output: 6
    }
}
```

# Key Points:



**Same Method Name:**

Multiple methods in the same class can have the same name if they have different parameter lists (different number of parameters or different types of parameters).

**Return Type:**

Method overloading can have the same or different return types, but the parameter list must differ.

**Access Modifier:**

Overloaded methods can have different access modifiers (e.g., public, protected, private).

**Compile-Time Polymorphism:**

Method overloading is a form of compile-time polymorphism, where the compiler determines which method to execute based on the method signature.

# Constructor in java



In Java, a constructor is a special method that is used to initialize objects when they are created.

Key points about constructors:

Name: The constructor has the same name as the class it belongs to.

Return type: It does not have a return type, not even void.

Purpose: It is used to initialize the object's state by assigning values to its member variables.

Automatic creation: If you don't define a constructor, the compiler automatically creates a default constructor with no arguments.

Overloading: You can create multiple constructors with different parameters, allowing you to initialize objects in different ways.

# Single inheritance



1. Single Inheritance:
A class inherits from only one parent class.
This is the simplest form of inheritance.

```
Class Animal {
    void eat() {
        System.out.println(“Animal is eating”);
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println(“Dog is barking”);
    }
}
```

# Lambda expression



A lambda expression in Java is a concise way to represent a functional interface (an interface with a single abstract method). It allows you to write code that is more compact and easier to read.

**Syntax**:

(parameter list) -> { body }

# Key words



**Parameter List:**

The list of parameters that the lambda expression accepts. If there are no parameters, use empty parentheses ().

**Arrow Token:**

The -> separates the parameter list from the body of the lambda expression.

**Body**:

The code that is executed when the lambda expression is invoked. It can be a single expression or a block of code enclosed in curly braces {}. If it's a block, you must use a return statement if the lambda expression needs to return a value.

**Common Use Cases:**

**Iterating over collections**

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.forEach(name -> System.out.println(name));
```

# Implementing functional interfaces.



```
Comparator<Integer> comparator = (x, y) -> Integer.compare(x, y);
```