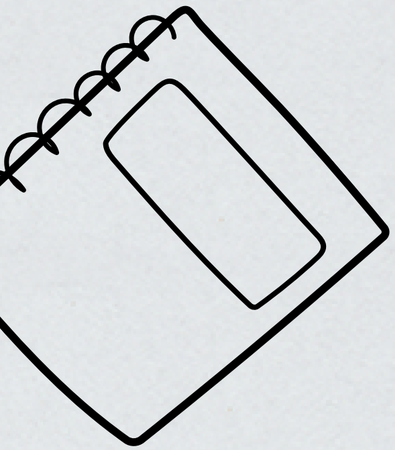
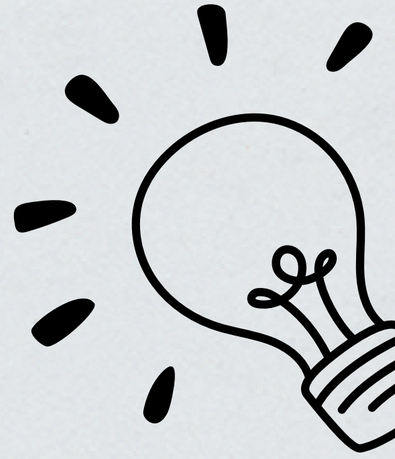


# TOPIC
# CLASSES, OBJECTS, METHODS, METHOD OVERLOADING




*Presented By :*

*Komal Pandey - 03104092023*

*Kratika Singh - 03204092023*

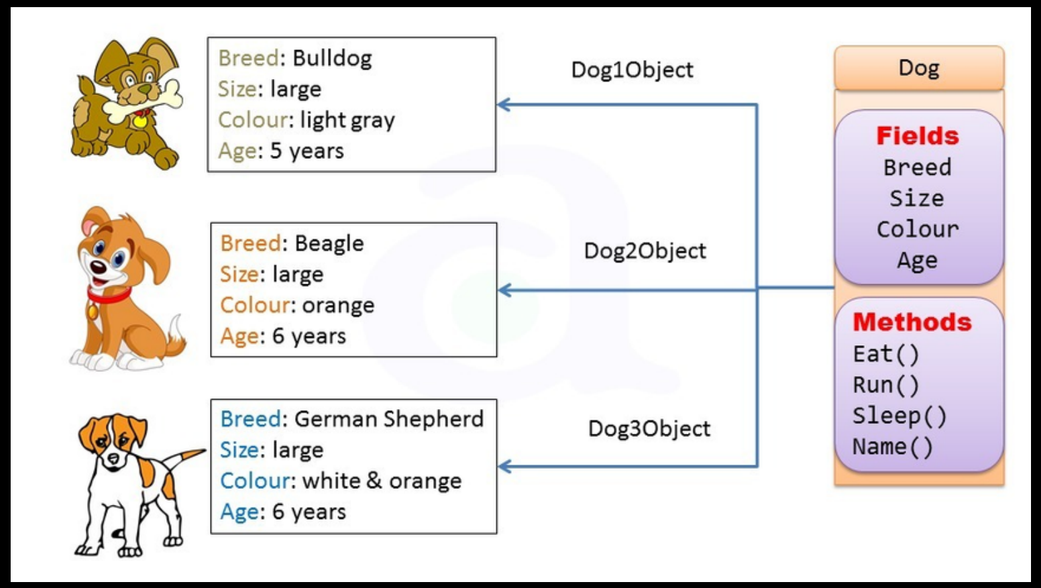
*Kritika Kukreja - 03304092023*

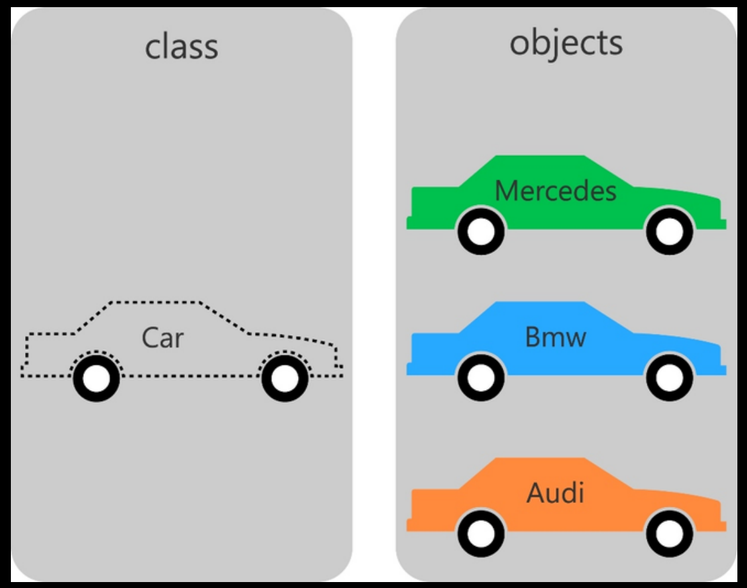
*Beena - 03404092023*

# What is a class?

- Java is an object-oriented programming language. The core concept of the object-oriented approach is to break complex problems into smaller objects.
- Everything in Java is associated with classes and objects, along with its attributes and methods.
- For example: in real life, a car is an object. The car has attributes, such as weight and color, and methods, such as drive, stop, accelerate etc.

A Class is like an **object constructor**, or a **"blueprint"** for creating objects.



# Components of a class

Let’s understand this using a real life example. Here, we’ve taken car as a class.

1. **Fields (Attributes): Represent the STATE of an object**
    - Brand: This could represent the brand or manufacturer of the car, such as "Toyota" or "Honda".
    - Model: This could represent the specific model of the car, such as "Camry" or "Civic".
    - Year: This could represent the manufacturing year of the car.

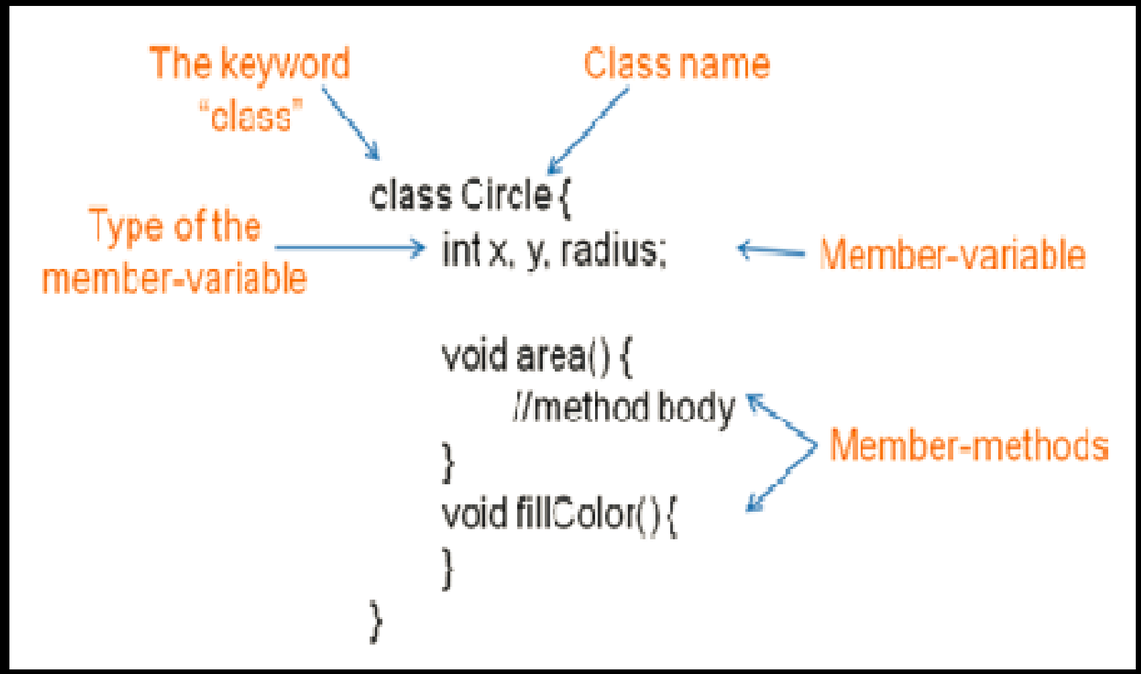
2. **Methods (Behaviors): Represent the BEHAVIOUR of an object**
    - Drive: This method would simulate the action of driving the car, where the car's engine runs, and it moves forward.
    - Stop: This method would simulate the action of stopping the car, where the car's engine stops, and it comes to a halt.
    - Accelerate, Brake, Turn: These are additional methods representing various actions a car can perform.

3. **Constructor:**
    - When a car is manufactured, it is given its initial brand, model, and year. This is similar to how a constructor initializes the fields of a class when an object is created.

4. **Access Modifiers:**
    - In real life, certain parts of a car may be accessible to the driver (public), while others are hidden from view and accessible only to mechanics (private).

# Declaration of a class

```
access_modifier class <class_name>
{
    data member;
    method;
    constructor;
    nested class;
    interface;
}
```



# Example of a class

```
class Student {
        // data member (also instance variable)
        int id;
        // data member (also instance variable)
        String name;

        public static void main(String args[])
       {
            // creating an object of
            // Student
            Student s1 = new Student();
            System.out.println(s1.id);
            System.out.println(s1.name);
    }
}
```

**OUTPUT**

0

null

# Here's a few more examples to get a clearer understanding of a class in Java.

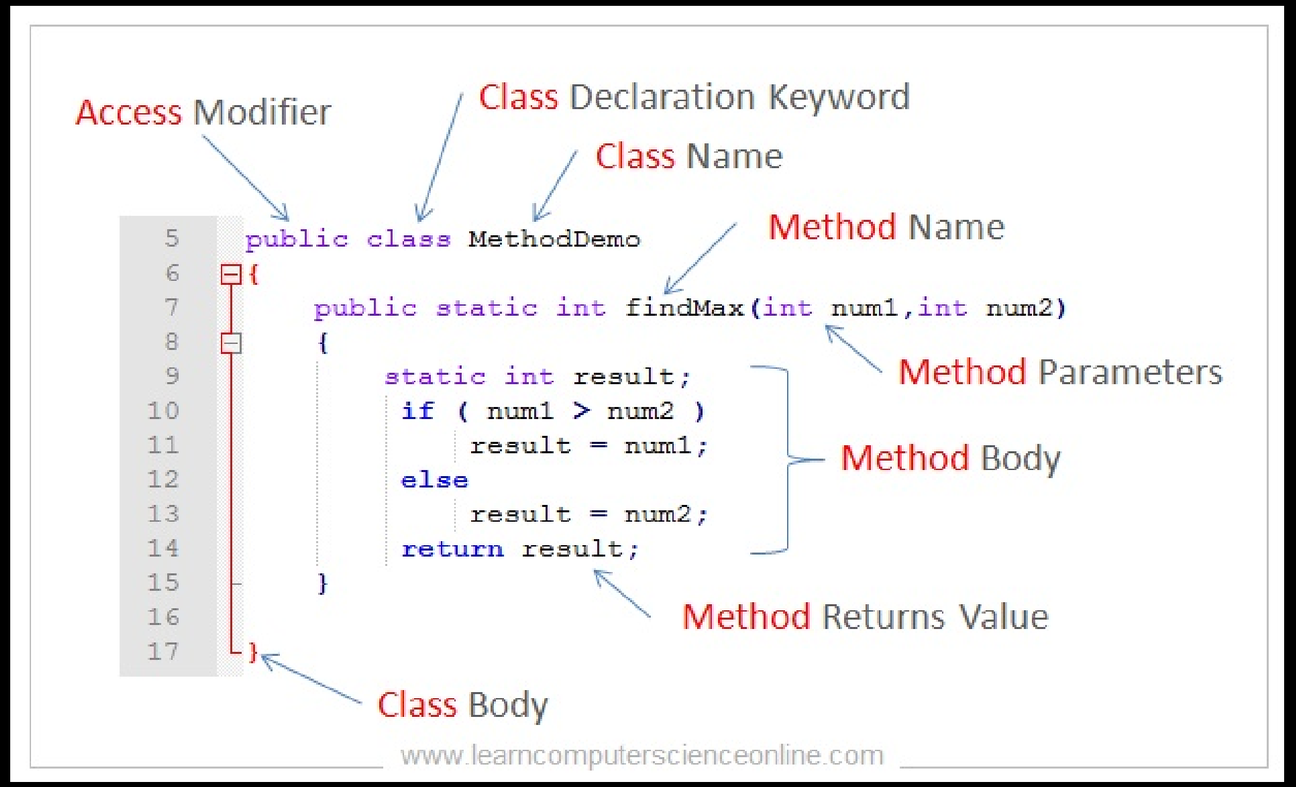
## An example of a class

```
class Person {
    String name;
    int age;

    void birthday ( ) {
        age++;
        System.out.println (name +
        ' is now ' + age);
    }
}
```

*Variable*

*Method*





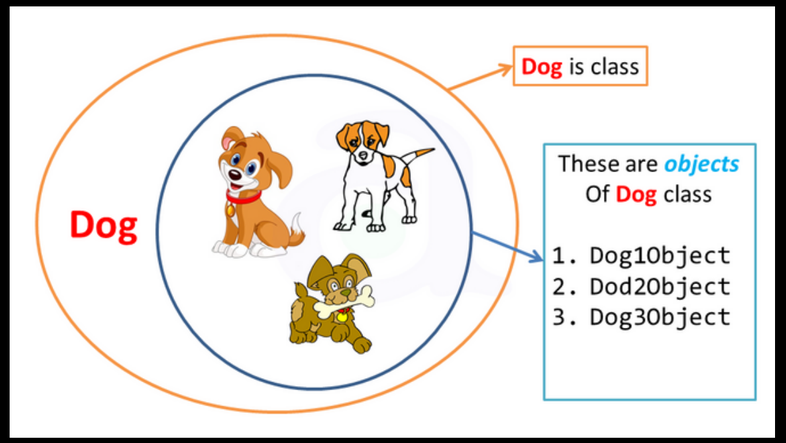

Here, we have created a class called Person where the age of the person will get incremented if birthday() method is called.

Here, we are finding out the maximum of 2 numbers inside a class.

# What is a Object?

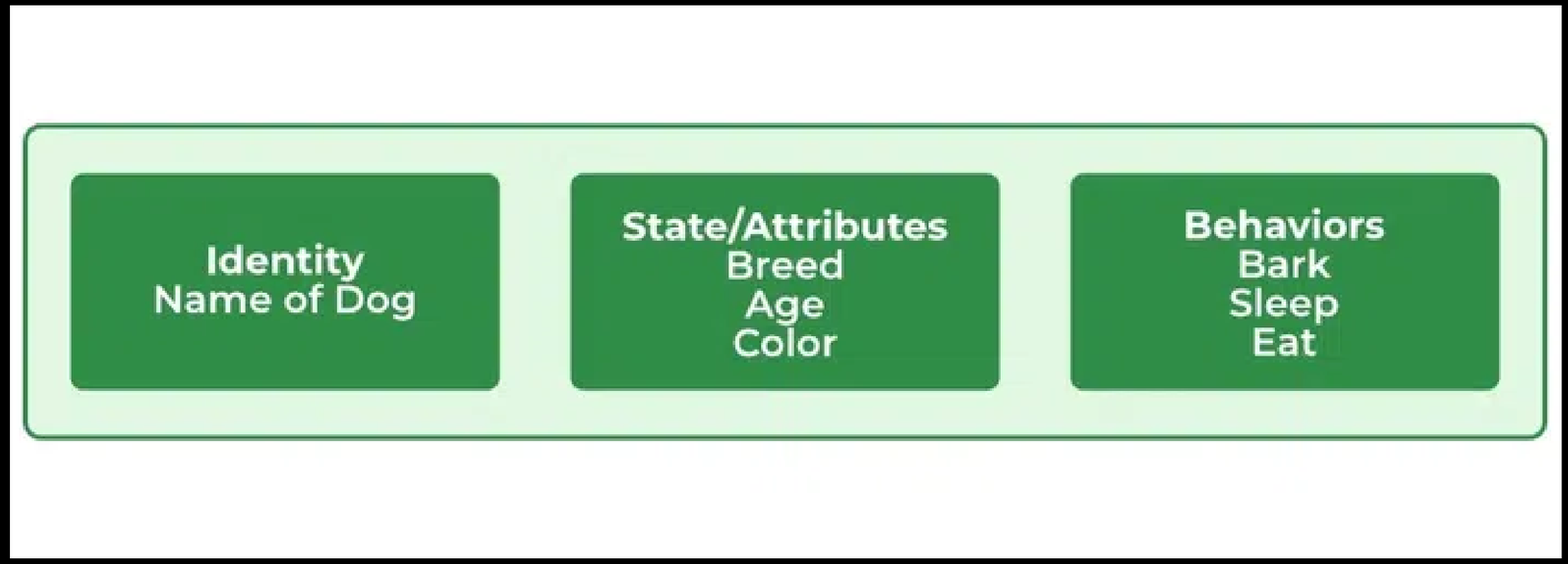
- An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities.
- Object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.
- An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.
- A typical Java program creates many objects, which as you know, interact by invoking methods

Example of an object: dog



# Components of a Object

1. **State:** It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior:** It is represented by the methods of an object. It also reflects the response of an object with other objects.
3. **Identity:** It gives a unique name to an object and enables one object to interact with other objects.



# Declaring Objects (Also called instantiating a class)

- When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class.
- But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

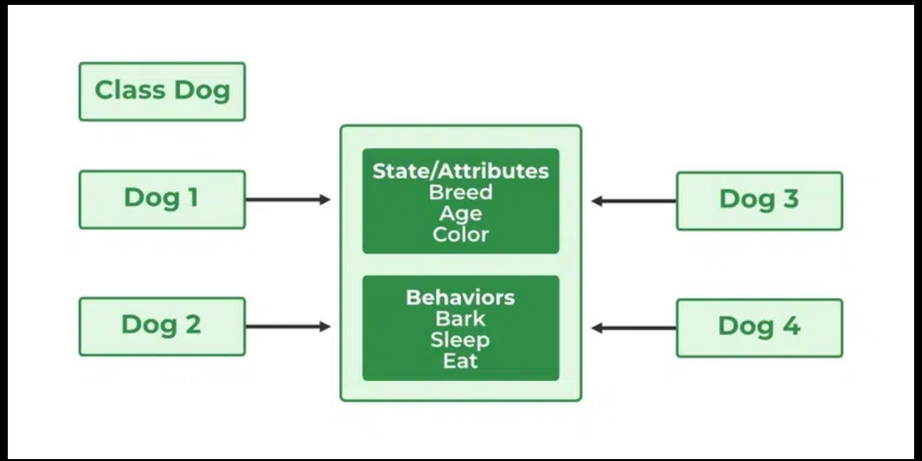
Example - Dog tuffy;

- If we declare a reference variable(tuffy) like this, its value will be undetermined(null) until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object.

# Initializing a Java object

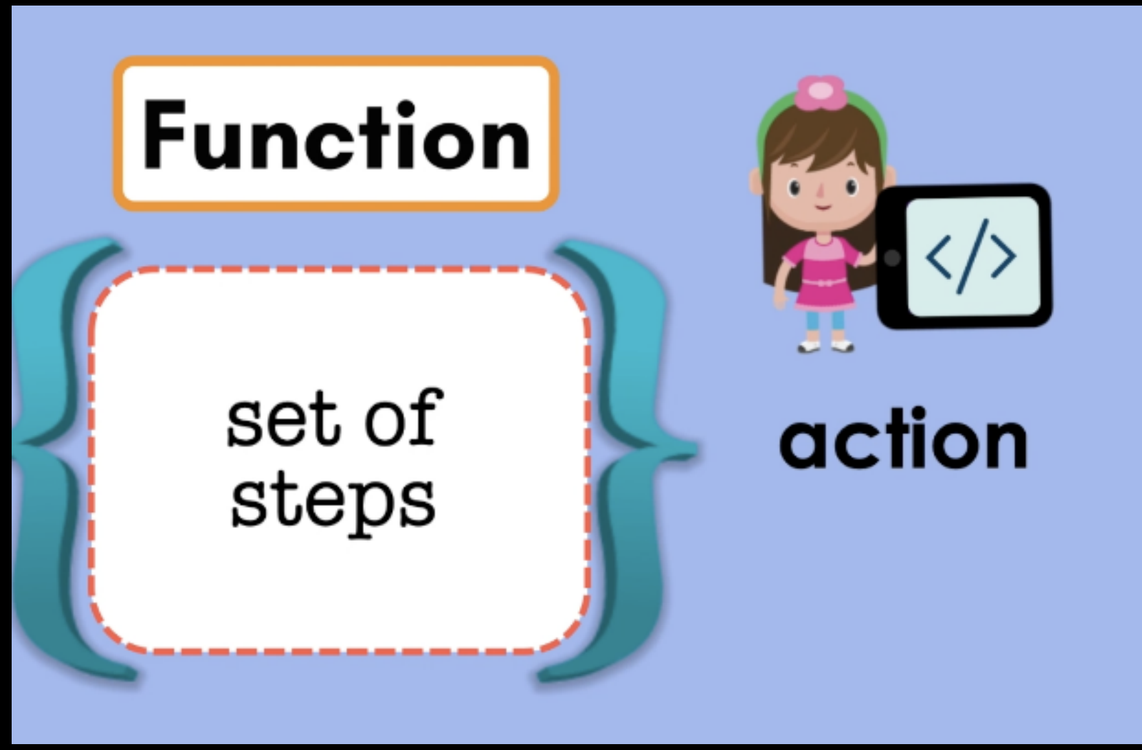
- The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.
- This reference is, essentially, the address in memory of the object allocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

```
Dog truffy; // declare reference to object
truffy = new Dog(); // allocate a Dog object
```



# Introduction to Methods

- Methods are self-contained blocks of code that perform specific tasks
- They are defined within a class and invoked using an object of that class
- Methods promote modularity, reusability, and code organization.



# Syntax

## Method Definition

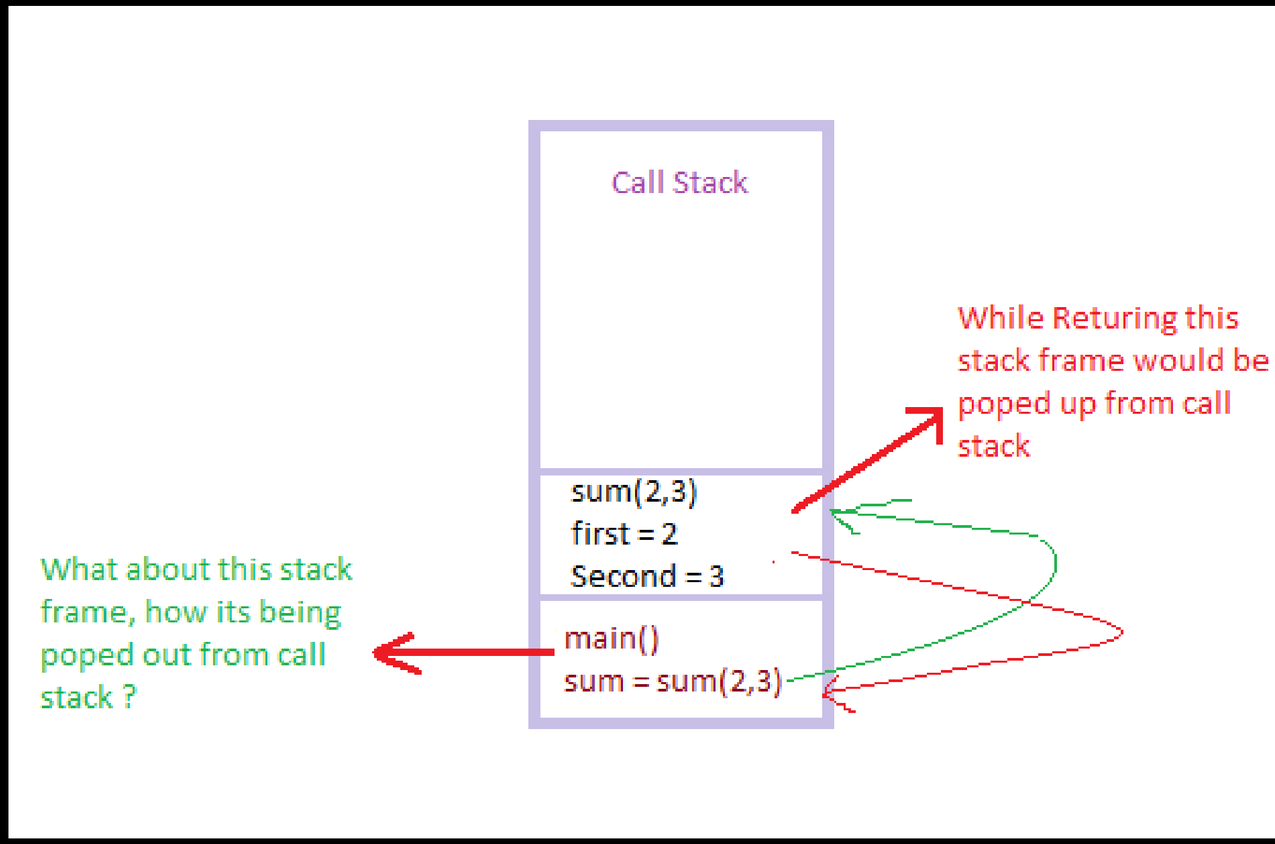
```
[accessSpecifier] [modifiers] returnType
methodName(parameterType1 parameter1, parameterType2
parameter2, ...) {
   // Method body
}
```

## Calling a Method

```
objectName.methodName(argument1, argument2);
```

# Invocation of a method

- Method Call
- Control Shift
- Execution Flow
- Return Handling



# Examples of a method

**Method with No Return Value:**

```
// Method to print a greeting message
public void greetUser(String name) {
 System.out.println("Hello, " + name + "!");
 }
```

**Method with Integer Return Type:**

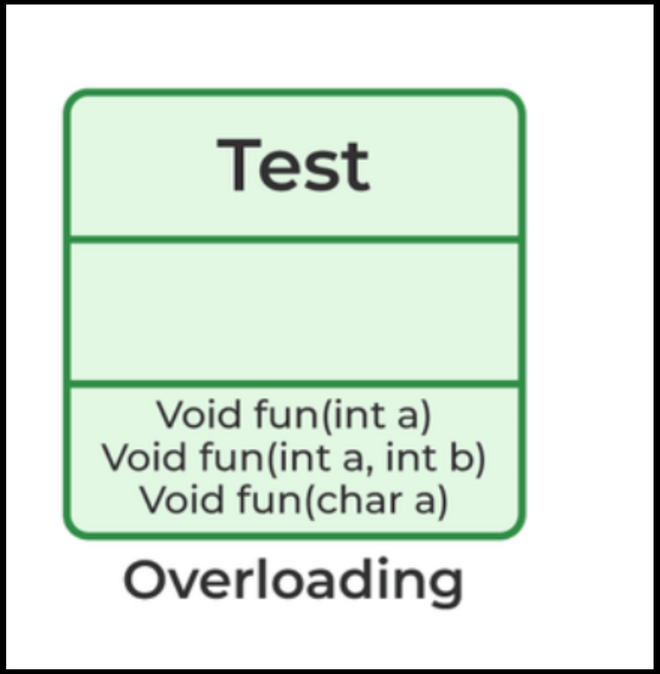
```
// Method to calculate the sum of two numbers
public int add(int num1, int num2) {
return num1 + num2;
}
```

## Method with Object Return Type:

```
// Method to return a new instance of a Person
public Person createPerson(String name, int age) {
Person p = new Person(name,age);
return p;
}
```

# METHOD OVERLOADING

- Method overloading in Java refers to the ability to define multiple methods in the same class with the same name but different parameters.



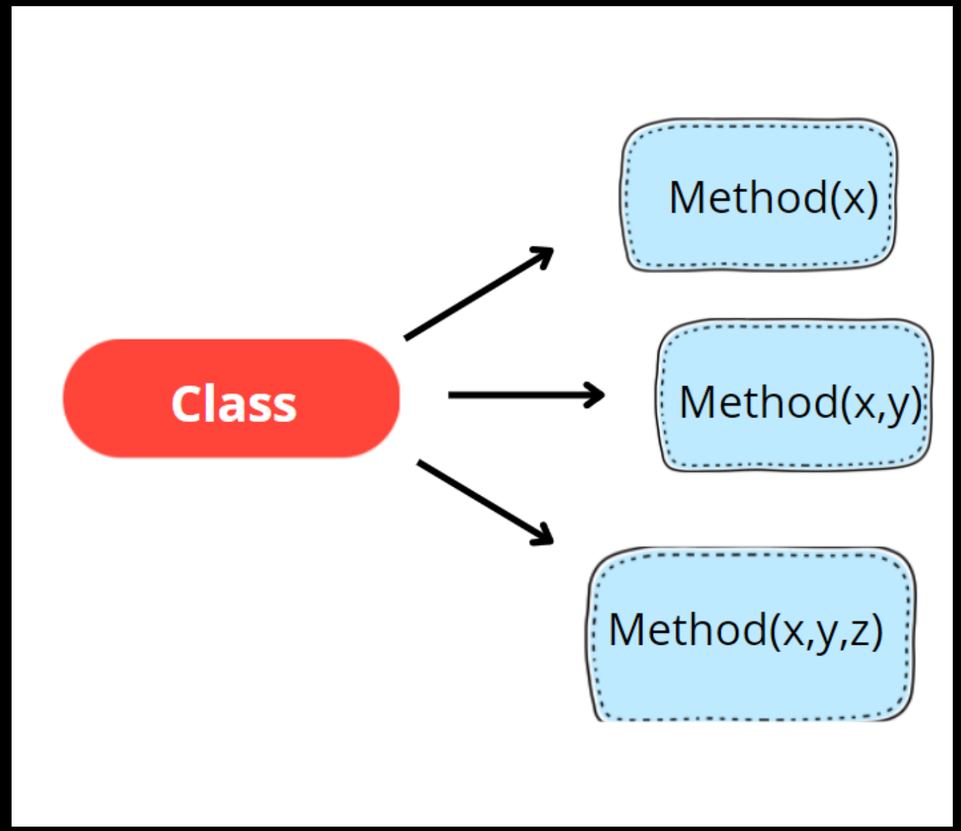

```
class Cat{

public void Sound(){
System.out.println("meow");
}

//overloading method
public void Sound(int num){
    for(int i=0; i<2;i++){
      System.out.println("meow");
     }
  }
}
```

OVERLOADING

Same method name but different parameters

# #Different Ways of Method Overloading

- Overloading by number of parameters.
- Overloading by data type of parameters.
- **Overloading by order of parameters**



# Overloading by number of parameters

- Methods with the same name but different numbers of parameters.

```java
public class OverloadingExample {
    // Method with no parameters
    public void display() {
        System.out.println("No parameters");
    }

    // Method with one parameter
    public void display(int num) {
        System.out.println("One parameter: " + num);
    }

    // Method with two parameters
    public void display(int num1, int num2) {
        System.out.println("Two parameters: " + num1 + ", " + num2);
    }

    public static void main(String[] args) {
        OverloadingExample example = new OverloadingExample();

        // Calling each overloaded method
        example.display();        // Calls display() with no parameters
        example.display(10);      // Calls display(int num)
        example.display(10, 20);  // Calls display(int num1, int num2)
    }
}
```

# Overloading by data type  of parameters

- Methods with the same name but parameters of different data types.

```java
public class OverloadingExample {
    public void print(int num) {
        System.out.println("Integer parameter: " + num);
    }

    public void print(double num) {
        System.out.println("Double parameter: " + num);
    }

    public static void main(String[] args) {
        OverloadingExample example = new OverloadingExample();
        example.print(10);       // Calls print(int num)
        example.print(10.5);     // Calls print(double num)
    }
}
```

# Overloading by order of parameters

- Methods with the same name but parameters arranged in a different order.

```java
public class OverloadingExample {
    public void print(int num, String str) {
        System.out.println("Int String: " + num + ", " + str);
    }

    public void print(String str, int num) {
        System.out.println("String Int: " + str + ", " + num);
    }

    public static void main(String[] args) {
        OverloadingExample example = new OverloadingExample();
        example.print(10, "Hello"); // Calls print(int num, String str)
        example.print("Hello", 10); // Calls print(String str, int num)
    }
}
```

# Why we need method overloading ?

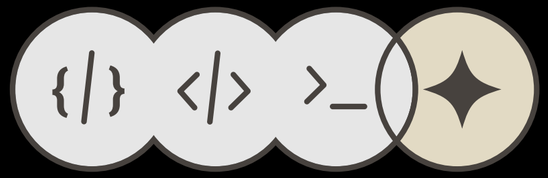
- **Readability:**
  - Simplifies code by using the same method name for related operations.
- **Reduced duplication:**
  - Avoids the need for multiple method names performing similar tasks.
- **Flexibility:**
  - Allows methods to accept different types or numbers of parameters.
- **Code reuse:**
  - Encourages reusing methods for similar tasks with different inputs.
- **Polymorphism:**
  - Supports polymorphic behavior, treating objects of different types uniformly.

**Without Method Overloading**

```
int add2(int x, int y)
{
    return(x+y);
}
int add3(int x, int y,int z)
{
    return(x+y+z);
}
int add4(int w, int x,int y, int z)
{
    return(w+x+y+z);
}
```

**With Method Overloading**

```
int add(int x, int y)
{
    return(x+y);
}
int add(int x, int y,int z)
{
    return(x+y+z);
}
int add(int w, int x,int y, int z)
{
    return(w+x+y+z);
}
```




Thank You!!