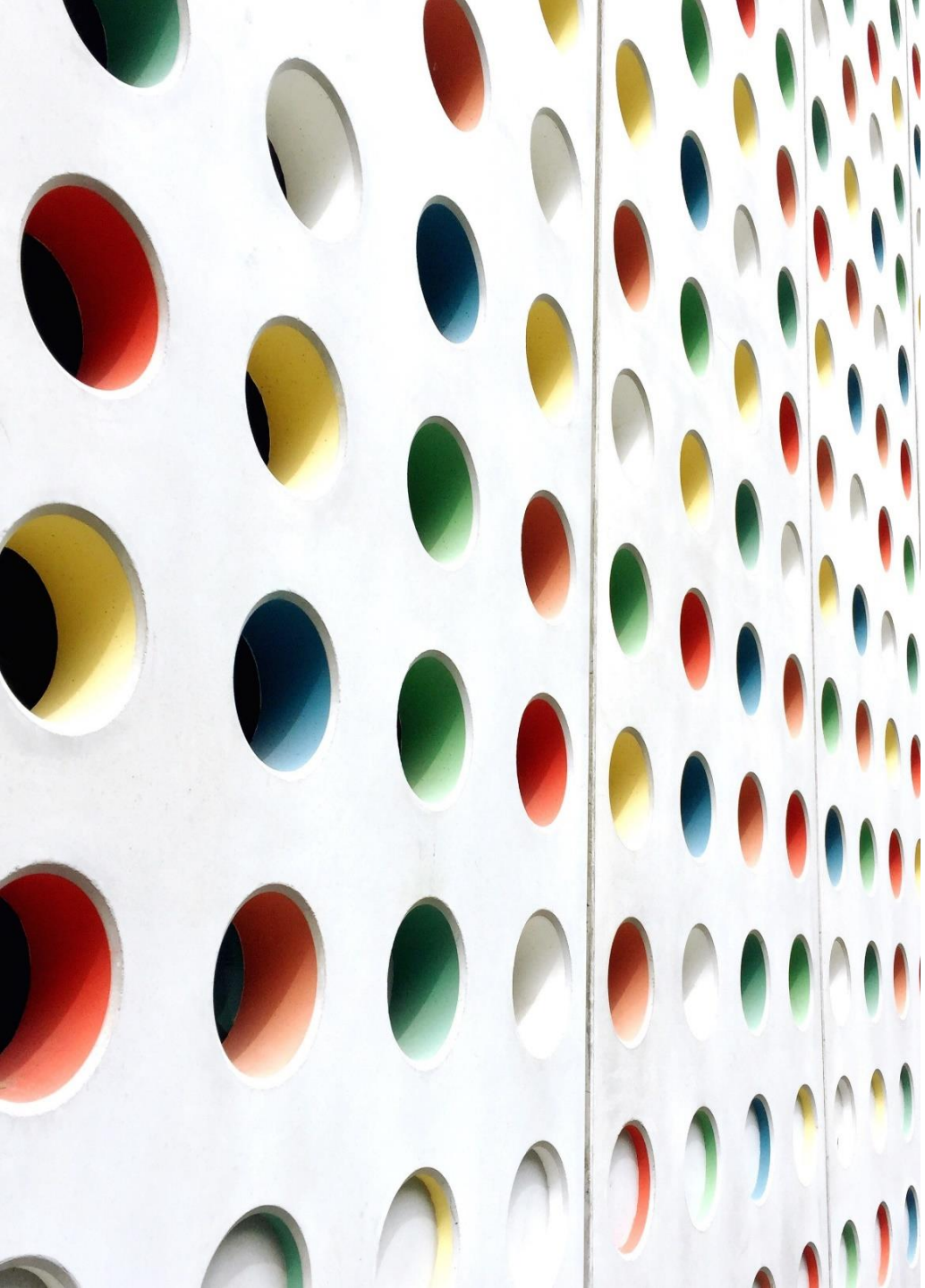


# *Chapter Two*

# Java Objects and Class

By: Sinodos G

## *Introduction*

- Java is an object-oriented programming language
- In object-oriented programming (OOP), programs are organized into **objects**
- The properties of objects are determined by their **class**
- Objects act on each other by passing **messages**
- There are some fundamental concepts of OOP

- ***Object***
- ***Class***
- ***Polymorphism***
- ***Inheritance***
- ***Encapsulation***
- ***Abstraction***
- ***Method***
- ***Message***

## Object

- A programming entity that contains state (data) and behavior (methods).
  - State: A set of values (internal data) stored in an object.
  - Behavior: A set of actions an object can perform, often reporting or modifying its internal state.
- Objects can be used as part of larger programs to solve problems.
- ***Example 1: Dogs***
  - States: name, color, breed, and "is hungry?"
  - Behaviors: bark, run, and wag tail
- ***Example 2: Cars***
  - States: color, model, speed, direction
  - Behaviors: accelerate, turn, change gears

## *Cont'd*

- An Object has two primary components:
  - ***state*** – properties of the object
  - ***behavior*** – operations the object can perform

State = Properties  Behavior = Method

**Constructing Objects**

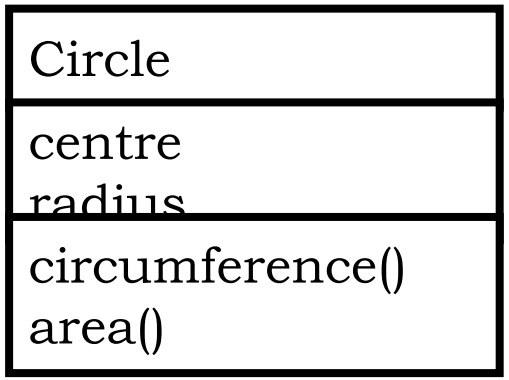
- use the **new** keyword to construct a new **instance** of an Object.
- assign this instance to a variable with the same type of the Object

- Note: classes define new datatypes !

```
LightSwitch ls = new LightSwitch();
```

## Class

- A class is the blueprint from which individual objects are created.
- A class defines a new data type which can be used to create objects of that type.
- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.
- Thus, a class is a template for an object, and an object is an instance of a class.
- Class= fields + methods



## *Classes and Objects*

- A Java program consists of one or more classes.
- A class is an abstract description of objects.
- Here is an example class:

```
class Dog {
    ...
    description of a dog goes here
    ...
}
```

- Here are some objects of that class:

## *Creating objects of a class*

- Object – block of memory that contains space to store all instance variables.
- Objects are created dynamically using the *new* keyword. It's called instantiating an object.
- Example:

```
class_name object_name;
object_name=new class_name();
                or
class_name object_name=new class_name();
```

- Object's created:
  - ObjectName.VariableName
  - ObjectName.MethodName(parameter-list)

## *Methods*

- a collection of statements that are grouped together to perform an operation.

### *Creating Method*

- ***Syntax***

```
public static int methodName(int a, int b) {
   // body
}
```

***Explanations:***

*public static – modifier*
*int – return type*
*methodName – name of the method*
*a, b – formal parameters*
*int a, int b – list of parameters*

**Method Calling**

- a method, it should be called for using
- There are two ways in which a method is called:-
  - method returns a value or
  - returning nothing (no return value).

- When a program invokes a method, the program control gets transferred to the called method. This called method then returns control to the caller in two conditions, when –
  - the return statement is executed.
  - it reaches the method ending closing brace.

- The methods returning void is considered as call to a statement.
- **Example:**

```
System.out.println("This is tutorialspoint.com!");
```

- The method returning value example −

```
int result = sum(6, 9);
```

- **The void Keyword**
  - The void keyword allows us to create methods which do not return a value.
  - The void method it which does not return any value.
  - Call to a void method must be a statement.

- **Passing Parameters by Value**
  - While working under calling process, arguments is to be passed.
  - These should be in the same order as their respective parameters in the method specification.
  - Parameters can be passed by value or by reference.
  - Passing Parameters by Value means calling a method with a parameter.
  - Through this, the argument value is passed to the parameter.

## Instance fields

- also known as instance variables.
- They are attributes or properties that belong to a specific instance (object) of a class.
- Each object created from a class can have its own set of values for these instance fields.

- ***Declaration:***
  - declared within a class but outside of any methods or constructors.
  - define the state of an object
  - marked with access modifiers like private, public, or protected to control their visibility and access.

***Example:***

```
public class Person {
    // Instance fields
    private String name;
    private int age;
}
```

## *Cont'd*

### *Initialization:*

- Instance fields can have default values if not explicitly initialized.

***Example***,

- numeric types are initialized to 0, and reference types (like objects) are initialized to null.

```
public class Person {
    private String name = "John";
    private int age = 30;
}
```

## *Cont'd*

- ***Access:***
  - Instance fields are accessed using the dot (.) notation
  - Each object has its own set of field values.

***Example:***

```
Person person1 = new Person();
    person1.name = "Alice";
    person1.age = 25;
Person person2 = new Person();
    person2.name = "Bob";
    person2.age = 40;
```

- ***Scope:***
  - Instance fields exist as long as the object they are associated with is in memory.
  - They are used to store and manage the object's data, and they are not shared across different instances of the class.

## Enumerated types

- referred to as enums,
- special data type used to define a set of constant values
- are a data type in computer programming that define a set of named constant values.
- powerful for creating well-structured code.
- used to represent a fixed set of related constants
- used to represent a finite, discrete set of options or choices.
- useful to improve code readability and maintainability

## Cont'd ...

- To create an enum, use the enum keyword
- Constants must be in uppercase letters
  - **Example**

```
enum Level {
    LOW,
    MEDIUM,
    HIGH
}
```

- can access enum constants with the dot
- ***Example:***
  - Level myVar = Level.MEDIUM;

- **Example:**

```
enum Color {
    RED,
    GREEN,
    BLUE
}
Color selectedColor =
Color.GREEN;
```

## *Cont'd . . .*

**Example:**

- write a java program tat check if today is Sunday using java enumerated.

```
enum Day {
        SUNDAY, MONDAY, TUESDAY,
        WEDNESDAY, THURSDAY,
        FRIDAY, SATURDAY
    }
public class EnumDay{
public static void main(String[] args) {
Day today = Day. SUNDAY;
if (today == Day. SUNDAY) {
        System.out.println("day is SUNDAY");
else
        System.out.println(" not Sunday ");
        }
    }
}
```

## *Cont'd*

- ***Enums*** are often used in switch statements to check for corresponding values:

```
enum Level {
  LOW,
  MEDIUM,
  HIGH
}
public class Main {
  public static void main(String[] args) {
    Level myVar = Level.MEDIUM;
    switch(myVar) {
      case LOW:
        System.out.println("Low level");
        break;
      case MEDIUM:
         System.out.println("Medium level");
        break;
      case HIGH:
        System.out.println("High level");
        break;
    }
  }
}
```

## *Enum and loops*

- The enum type has a values() method, which returns an array of all enum constants.
- This method is useful when you want to loop through the constants of an enum:
- **Example:**

```
enum Level {
  LOW,  MEDIUM,   HIGH
}
}
public class Main {
  public static void main(String[] args) {
    for (Level myVar : Level.values()) {
      System.out.println(myVar);
    }
  }
```

## *Constructors*

- a **special method** that is used to initialize objects.
- called when an object of a class is created.
- It can be used to set initial values for object attributes
- fundamental building blocks of classes
- Constructor is invoked at the time of object creation.

- ***Purpose:***
  - are special methods used to create and initialize objects of a class.
  - called when an object is instantiated (created) using the new keyword.

- ***Naming:***
  - have the same name as the class, and they do not have a return type, not even void.

## *Cont'd*

**Rules for creating constructor**

- **Constructor name must be same as its class name**
  - ❑ **Constructor Name=Class Name**
- **Constructor must have no explicit return type**

***Example:***

```
public class Person {
    // Constructor

 public Person() {
        // Initialization code goes here
    }
}
```

## *Cont'd*

- **Types of constructors**
  - Default Constructor
  - Parameterized Constructor
  - *Copy Constructor:*
    - Creates a new object by copying the values from an existing object.
  - *Static Constructor*
  - *Private Constructor:*
    - Used to prevent the instantiation of a class, often in singleton design patterns.

## *Cont'd*

### Types of Constructors

- **Default constructor (No-argument constructor)**
  - A constructor that have no parameter
  - Default values to the object like 0, null etc. depending on the type.

```
<class_name>()
    {
    }
```

- **Parameterized constructor**
  - A constructor that have parameters
  - Used to provide different values to the distinct objects.

```
<class_name>(parameter list)
    {
    }
```

## *Example*

```
class Student {
      int id;
      String name;
void display()
            {
System.out.println(id+" "+name);
            }
public static void main(String args[]) {
      Student s1=new Student();
      Student s2=new Student();
      s1.display();
      s2.display();
            }
}
```

```
class_name object_name=new class_name();
```

## Example:

```
class Student{
int id;
String name;
Student(int i,String n){
id = i;
name = n;
}

void display(){
System.out.println(id+" "+name);
}
```

```
public static void main(String args[]){
Student s1 = new Student(111,“Abay");

Student s2 = new Student(222,“Gebissa");

 s1.display();
s2.display();
}
}
```

## *Cont'd*

***Example:***

```
public class Main {
  int x;
  public Main() {
    x = 5;  // Set the initial value for the class attribute x
  }
  public static void main(String[] args) {
    Main aa = new Main();
// Create an object of class Main (This will call the constructor)
    System.out.println(aa.x);
// Print the value of x
  }}
```

## *Cont'd*

- ***Example: Constructor wit parameters***

```
public class Main {
  int x;
  public Main(int y) {
    x = y;
  }
  public static void main(String[] args) {
    Main aa = new Main(5);
    System.out.println(aa.x);
  }
}
```

# Java - Methods

- a collection of statements that are grouped together to perform an operation. Example:
  - When you call the System.out.**println()** method, the system actually executes several statements in order to display a message on the console.
- ***Creating Method***
  - Considering the following example to explain the syntax of a method

  - ***Syntax***

```
public static int methodName(int a, int b) {
   // body
}
```

## ❑ *Constructor Overloading*

- a technique in which a class can have any number of constructors that differ in parameter lists.
- The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

## ❑ *Method Overloading*

- class have multiple methods by the same name but different parameters, it is known as Method Overloading.

## *Example: Constructor Overloading*

```
class Student{
    int id;
    String name;
    int age;
Student(int i,String n){
    id = i;
    name = n;
    }
Student(int i,String n,int a){
    id = i;
    name = n;
    age=a;
}
void display(){
System.out.println(id+" "+name+" "+age);
}
public static void main(String args[]){
    Student s1 = new
    Student(111,“yemane");
    Student s2 = new
    Student(222,"Aryan",25);
    s1.display();
    s2.display();
  }
}
```

## Methods

- a block of code which only runs when it is called.
- You can pass data, known as parameters, into a method.
- used to perform certain actions, and they are also known as **functions**.
- Objects interact with each other by passing **messages.**
- **Purpose**
  - functions defined within a class to perform specific tasks or operations.
  - They define the behavior of the class and enable you to encapsulate functionality.
- **Naming**
  - Methods have names that are relevant to the action they perform.
  - They can have various return types, including void for methods that do not return a value.

## *Cont'd*

```
public class Calculator {
    // Method to add two numbers and return the result
    public int add(int num1, int num2) {
        return num1 + num2;
    }

    // Method with no return value
    public void dispaly(String message) {
        System.out.println(message);
    }
}
```

## *Cont'd*

### *Example: Methods with parameters*

```
public class Main {
    static void myName(String fname) {
        System.out.println(fname + " Sinodos");
    }
public static void main(String[] args) {
    Name("Abel");
    Name("Marta");
    Name("Yonas");
  }
}
```

## *Method Overloading*

- Refer to the multiple methods with the same name in a class, as long as they have different parameter lists.

```
public class Calculator {
        public int add(int num1, int num2) {
            return num1 + num2;
        }

    public double add(double num1, double num2)
        {
            return num1 + num2;
        }
}
```

## *Cont'd*

***Example:***

```
static int add(int x, int y) {
  return x + y;
}
static double add(double x, double y) {
  return x + y;
}
public static void main(String[] args) {
  int num1 = add(8, 5);
  double num2 = add(4.3, 6.26);
  System.out.println("int: " + num1);
  System.out.println("double: " + num2);
}
```

## *Example of Method*

```
public class AddIntegers {
        public static void main(String[] args) {
            int num1 = 5;
            int num2 = 7;
            int sum = addNumbers(num1, num2);
            System.out.println("Sum: " + sum);
        }

        public static int addNumbers(int a, int b) {
            return a + b;
        }
    }
```

# Example – Constructor

```
public class AddIntegers {
    private int num1;
    private int num2;

    // Constructor to initialize the two integers
    public AddIntegers(int num1, int num2) {
        this.num1 = num1;
        this.num2 = num2;
    }

    // Method to add the two integers
    public int add() {
        return num1 + num2;
    }
```

```
public static void main(String[] args) {
        // Create an object of the AddIntegers class and pass the
two integers to the constructor
        AddIntegers adder = new AddIntegers(5, 7);

        // Call the add() method to get the result
        int sum = adder.add();

        System.out.println("The sum of the two integers is: " +
sum);
    }
}
```

## *Access Modifiers*

- are keywords that determine the visibility or accessibility of classes, methods, fields, and other members within a program.
- There are four main access modifiers in Java:
  - public: Accessible from anywhere.
  - protected: Accessible within the same package and in subclasses.
  - default: Accessible only within the same package.
  - private: Accessible only within the same class.

- Additionally, Java provides two more access modifiers for classes:
  - final: A final class cannot be extended (subclassed).
  - abstract: An abstract class cannot be instantiated on its own and is meant to be subclassed.

## *Cont'd*

- ***Example:***

```
public class Numbers {
  final int x = 10;
  final double pi = 3.14;
  public static void main(String[] args) {
    Numbers aa = new Numbers();
    aa.x = 50;
    aa.PI = 25;
System.out.println(aa.x);
System.out.println(pi.x);
  }
}
```

Find Output?

## *Cont'd*

```
public class Main {
static void myStaticMethod() {
    System.out.println("Static methods . . .");
  }
  // Public method
  public void myPublicMethod() {
    System.out.println("Public methods . . .");
  }
        public static void main(String[ ] args) {
            myStaticMethod(); // Call the static method
            // myPublicMethod(); error

            Main aa = new Main(); // Create an object of Main
            aa.myPublicMethod(); // Call the public method
          }
        }
```

- A static method means that it can be accessed without creating an object of the class, unlike public:

## *Cont'd*

- Difference between constructor and method

### Constructor

- Constructor is used to initialize the state of an object.
- Constructor must not have return type.
- Constructor is invoked implicitly.
- The java compiler provides a default constructor if you don't have any constructor.
- Constructor name must be same as the class name.

### Method

- Method is used to expose behaviour of an object.
- Method must have return type.
- Method is invoked explicitly.
- Method is not provided by compiler in any case.
- Method name may or may not be same as class name.

## *Encapsulation*

- A fundamental principles of object-oriented programming in Java.
- Means make sure that "sensitive" data is hidden from users.
  - declare class variables/attributes as private
  - provide public get and set methods to access and update the value of a private variable
- Bundling of data (attributes or fields) and methods (functions) that operate on that data into a single unit, known as a class.
- The class is typically declared as private, and access to that data is controlled through public methods, which are called getters and setters.
- This ensures that the data is only accessed and modified in a controlled and consistent manner, promoting data integrity and security.

## *Cont'd*

- Encapsulation is one of the four fundamental OOP concepts.
- The other three are inheritance, polymorphism, and abstraction.
- a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.
- Variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.
- To achieve encapsulation in Java –
  - Declare the variables of a class as private.
  - Provide public setter and getter methods to modify and view the variables values.

## *Example*

```
public class EncapTest {
    private String name;
    private String idNum;
    private int age;
public int getAge() {
        return age; }
public String getName() {
        return name; }
public String getIdNum() {
        return idNum; }
```

```
public void setAge( int newAge) {
age = newAge; }
 public void setName(String newName) {
 name = newName; }
public void setIdNum( String newId) {
idNum = newId;
} }
```

## *Cont'd*

- The public setXXX() and getXXX() methods are the access points of the instance variables of the EncapTest class.
- Normally, these methods are referred as getters and setters. Therefore, any class that wants to access the variables should access them through these getters and setters.
- The variables of the EncapTest class can be accessed using the following program:

```
public class RunEncap {
public static void main(String args[]) {
EncapTest encap = new EncapTest();
encap.setName("James");
encap.setAge(20);
encap.setIdNum("12343ms");
System.out.print("Name : " + encap.getName() + " Age : " + encap.getAge()); } }
```

## *Cont'd*

- ***Key aspects of encapsulation***
  - Data Hiding
  - Public Methods
  - Access Control:
    - Defines which methods and data members are accessible to the outside world and which are kept private.
  - Information Abstraction:
    - abstracts the implementation details of a class and focuses on what the class can do rather than how it does it.

## *Benefits of encapsulation*

- **Data Protection**
  - prevent accidental or unauthorized modifications
  - The fields of a class can be made read-only or write-only.
  - A class can have total control over what is stored in its fields.
- **Flexibility**
  - allows you to change the internal implementation of a class without affecting the code that uses the class
- ***Code Maintainability***
  - It promotes code organization and makes it easier to understand and maintain by providing a clear and well-defined structure.
- **Reusability**
  - used as building blocks in different parts of a program or in different programs altogether, enhancing code reusability.

## *Example*

```
public class Student {
    // Private fields (data)
    private String name;
    private int age;

    // Public constructor
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Public getter method for name
        public String getName() {
            return name;
        }

    // Public setter method for name
        public void setName(String name) {
            this.name = name;
        }

    // Public getter method for age
        public int getAge() {
            return age;
        }

    // Public setter method for age
        public void setAge(int age) {
            if (age >= 0) {
                this.age = age;
            }
        }
}
```

# Question

End of Chapter 2

Next → Chapter 3