# SNS COLLEGE OF TECHNOLOGY

**Coimbatore-35**
**An Autonomous Institution**
Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A+' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

## DEPARTMENT OF AIML

## OBJECT ORIENTED PROGRAMMING USING JAVA

UNIT 2 – Basics Features Of Java

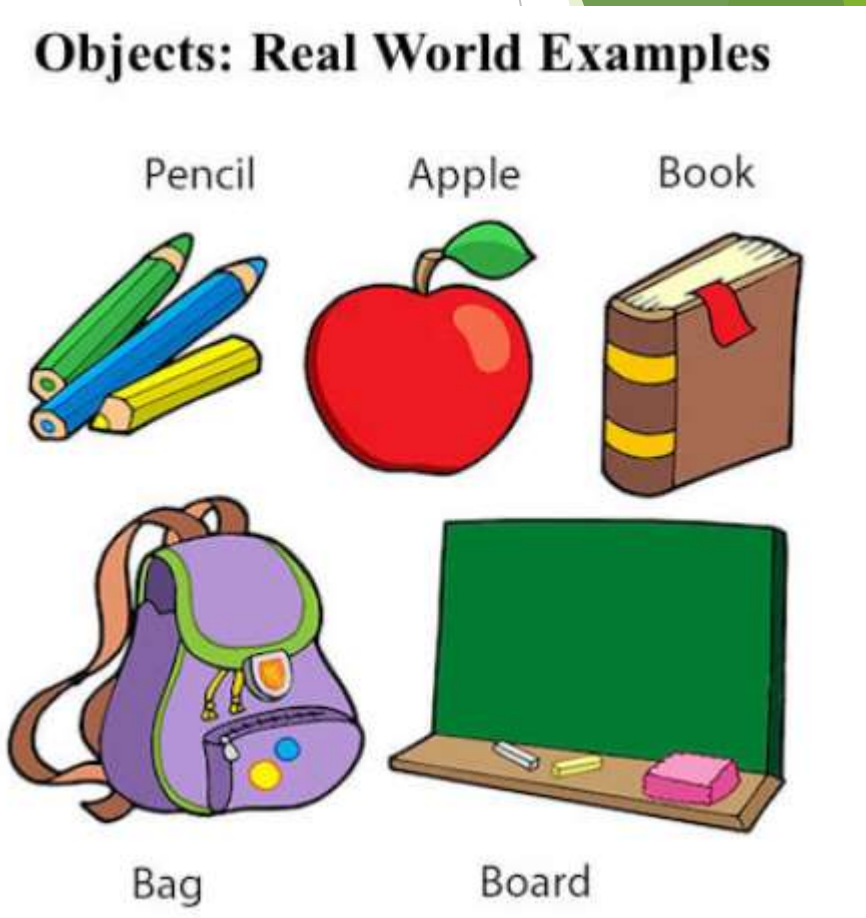
Topic 3 – Objects, Class & Methods

# Objects and Classes in Java

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

## What is an object in Java?

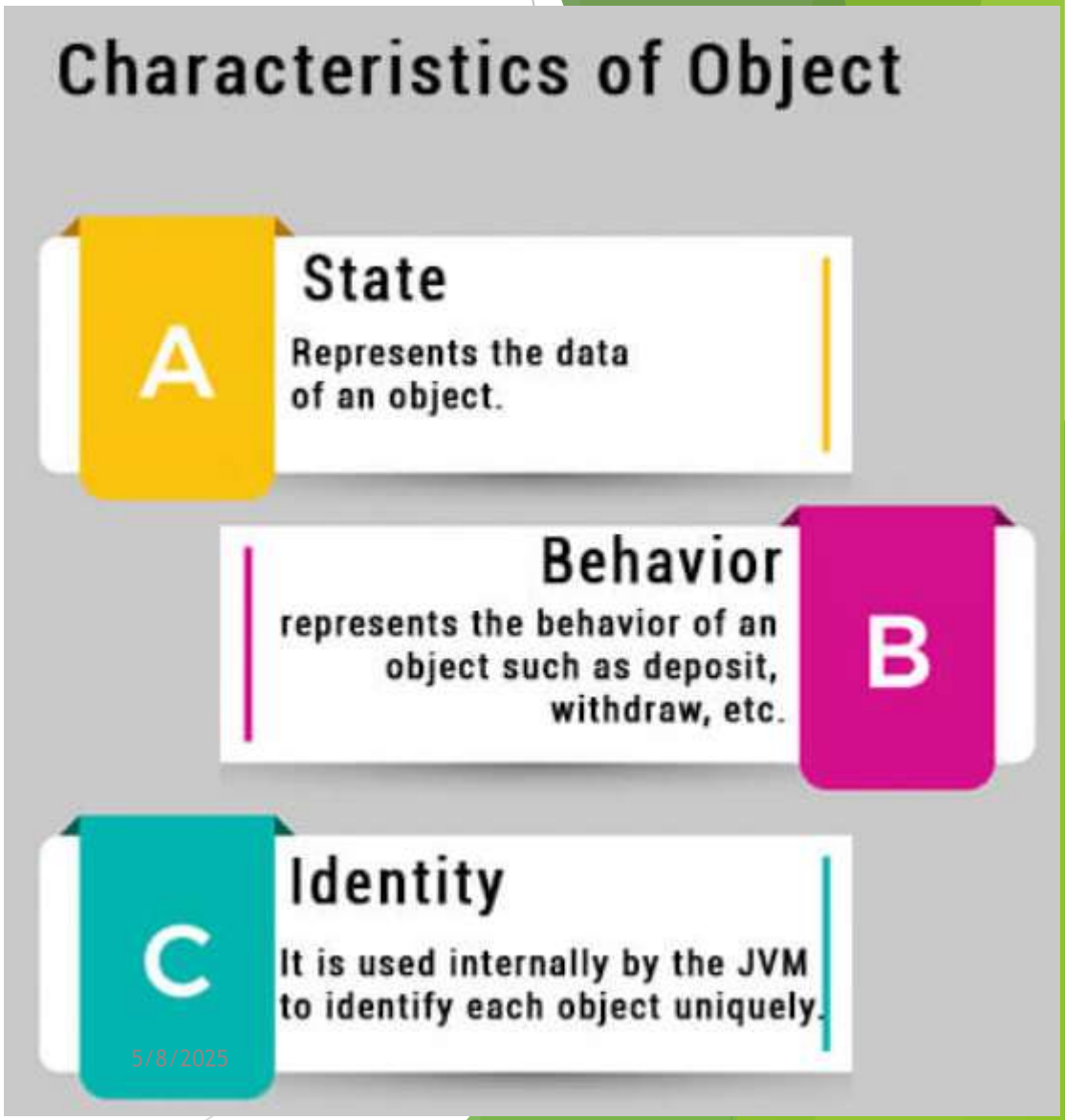
- An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc.
- It can be physical or logical (tangible and intangible).
- The example of an intangible object is the banking system.



# Objects and Classes in Java

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.
- For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.



# Objects and Classes in Java

**An 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.

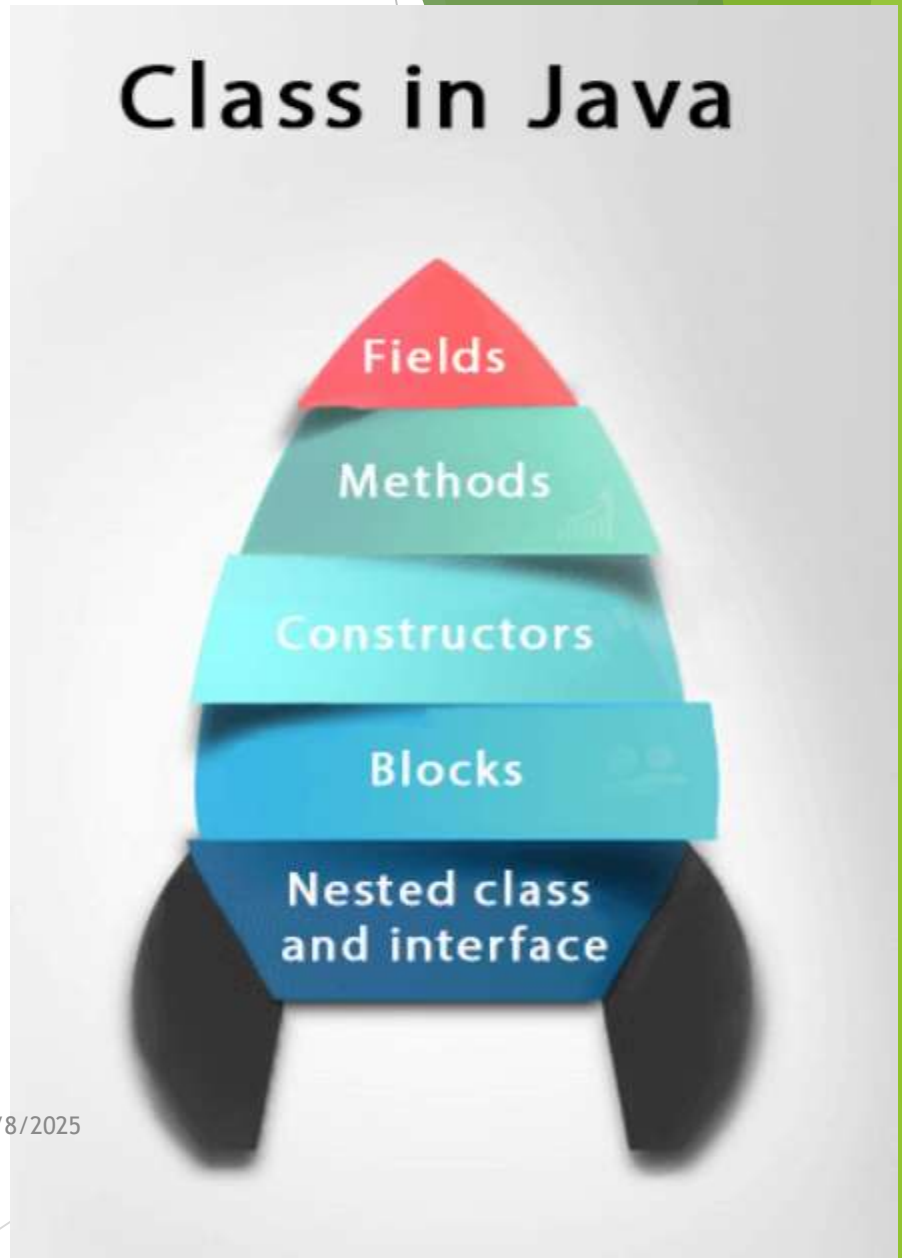
**Object Definitions:**

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

# Objects and Classes in Java

## What is a class in Java?

- A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
- A class in Java can contain:
  - **Fields**
  - **Methods**
  - **Constructors**
  - **Blocks**
  - **Nested class and interface**



# Objects and Classes in Java

## Defining a Class in Java

Syntax:

```
public class class_name
{
    Data Members;
    Methods;
}
```

Example:

```
public class Car
{
    public:
    double color; // Color of a car
    double model; // Model of a car
}
```

- Private , Protected , Public is called visibility labels.
- The members that are declared private can be accessed only from within the class.
- Public members can be accessed from outside the class also.

# Objects and Classes in Java

## Class Members

`Data` and `functions` are members.

Data Members and methods must be declared within the class definition.

Example:

```
public class Cube
{
    int length; // length is a data member of class Cube
    int breadth; // breadth is a data member of class Cube
    int length ; // Error redefinition of length
}
```

- A member cannot be redeclared within a class.
- No member can be added elsewhere other than in the class definition.

# Objects and Classes in Java

## Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

## new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

# Objects and Classes in Java

## Create a Class

To create a class, use the keyword `class`:

### Main.java

Create a class named "`Main`" with a variable x:

```
public class Main {
  int x = 5;
}
```

## Create an Object

### Example

Create an object called "`myObj`" and print the value of x:

```
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

# Objects and Classes in Java

## Multiple Objects

You can create multiple objects of one class:

### Example

Create two objects of `Main`:

```
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj1 = new Main();  // Object 1
    Main myObj2 = new Main();  // Object 2
    System.out.println(myObj1.x);
    System.out.println(myObj2.x);
  }
}
```

# Objects and Classes in Java

## Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the `main()` method (code to be executed)).

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory/folder:

- Main.java
- Second.java

Main.java

```
public class Main {
  int x = 5;
}
```

Second.java

```
class Second {
  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

# Objects and Classes in Java

When both files have been compiled:

```
C:\Users\Your Name>javac Main.java
C:\Users\Your Name>javac Second.java
```

Run the Second.java file:

```
C:\Users\Your Name>java Second
```

And the output will be:

```
5
```

# Objects and Classes in Java

## Java Class Attributes

We used the term "variable" for `x` in the example (as shown below). It is actually an **attribute** of the class. Or you could say that class attributes are variables within a class:

### Example

Create a class called "`Main`" with two attributes: `x` and `y`:

```
public class Main {
  int x = 5;
  int y = 3;
}
```

Another term for class attributes is **fields**.

# Objects and Classes in Java

## Accessing Attributes

You can access attributes by creating an object of the class, and by using the dot syntax ( `.` ):

The following example will create an object of the `Main` class, with the name `myObj`. We use the `x` attribute on the object to print its value:

### Example

Create an object called " `myObj` " and print the value of `x` :

```
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println(myObj.x);
  }
}
```

# Objects and Classes in Java

## Modify Attributes

You can also modify attribute values:

### Example

Set the value of `x` to 40:

```
public class Main {
  int x;

  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 40;
    System.out.println(myObj.x);
  }
}
```

Or override existing values:

### Example

Change the value of `x` to 25:

```
public class Main {
  int x = 10;

  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 25; // x is now 25
    System.out.println(myObj.x);
  }
}
```

# Objects and Classes in Java

If you don't want the ability to override existing values, declare the attribute as `final`:

## Example

```
public class Main {
  final int x = 10;

  public static void main(String[] args) {
    Main myObj = new Main();
    myObj.x = 25; // will generate an error: cannot assign a value to a final variable
    System.out.println(myObj.x);
  }
}
```

The `final` keyword is useful when you want a variable to always store the same value, like PI (3.14159...).



The `final` keyword is called a "modifier".

# Objects and Classes in Java

## Multiple Objects

If you create multiple objects of one class, you can change the attribute values in one object, without affecting the attribute values in the other:

### Example

Change the value of `x` to 25 in `myObj2`, and leave `x` in `myObj1` unchanged:

```
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj1 = new Main();  // Object 1
    Main myObj2 = new Main();  // Object 2
    myObj2.x = 25;
    System.out.println(myObj1.x);  // Outputs 5
    System.out.println(myObj2.x);  // Outputs 25
  }
}
```

# Objects and Classes in Java

## Multiple Attributes

You can specify as many attributes as you want:

### Example

```
public class Main {
  String fname = "John";
  String lname = "Doe";
  int age = 24;

  public static void main(String[] args) {
    Main myObj = new Main();
    System.out.println("Name: " + myObj.fname + " " + myObj.lname);
    System.out.println("Age: " + myObj.age);
  }
}
```

# Methods in Java

## Java 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**.

Why use methods? To reuse code: define the code once, and use it many times.

## 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:

# Methods in Java

## Example

Create a method inside Main:

```
public class Main {
  static void myMethod() {
    // code to be executed
  }
}
```

## Example Explained

- `myMethod()` is the name of the method
- `static` means that the method belongs to the Main class and not an object of the Main class. You will learn more about objects and how to access methods through objects later in this tutorial.
- `void` means that this method does not have a return value. You will learn more about return values later in this chapter

# Methods in Java

## Call a Method

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

In the following example, `myMethod()` is used to print a text (the action), when it is called:

### Example

Inside `main`, call the `myMethod()` method:

```
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!"
```

# Methods in Java

A method can also be called multiple times:

## Example

```
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!
```

`String[] args` means an array of sequence of characters (Strings) that are passed to the "main" function. This happens when a program is executed.

Example when you execute a Java program via the command line:

```
java MyProgram This is just a test
```

Therefore, the array will store: `["This", "is", "just", "a", "test"]`

# Java Method Parameters

## Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a `String` called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

# Java Method Parameters

## Example

```
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
```

When a **parameter** is passed to the method, it is called an **argument**. So, from the example above: `fname` is a **parameter**, while `Liam`, `Jenny` and `Anja` are **arguments**.

# Java Method Parameters

## Multiple Parameters

You can have as many parameters as you like:

Example

```
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
```

Note that when you are working with multiple parameters, the method call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

# Java Method Parameters

## 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:

### Example

```
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)
```

# Java Method Parameters

This example returns the sum of a method's **two parameters**:

## Example

```
public class Main {
  static int myMethod(int x, int y) {
    return x + y;
  }

  public static void main(String[] args) {
    System.out.println(myMethod(5, 3));
  }
}
// Outputs 8 (5 + 3)
```

# Java Method Parameters

You can also store the result in a variable (recommended, as it is easier to read and maintain):

## Example

```
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)
```