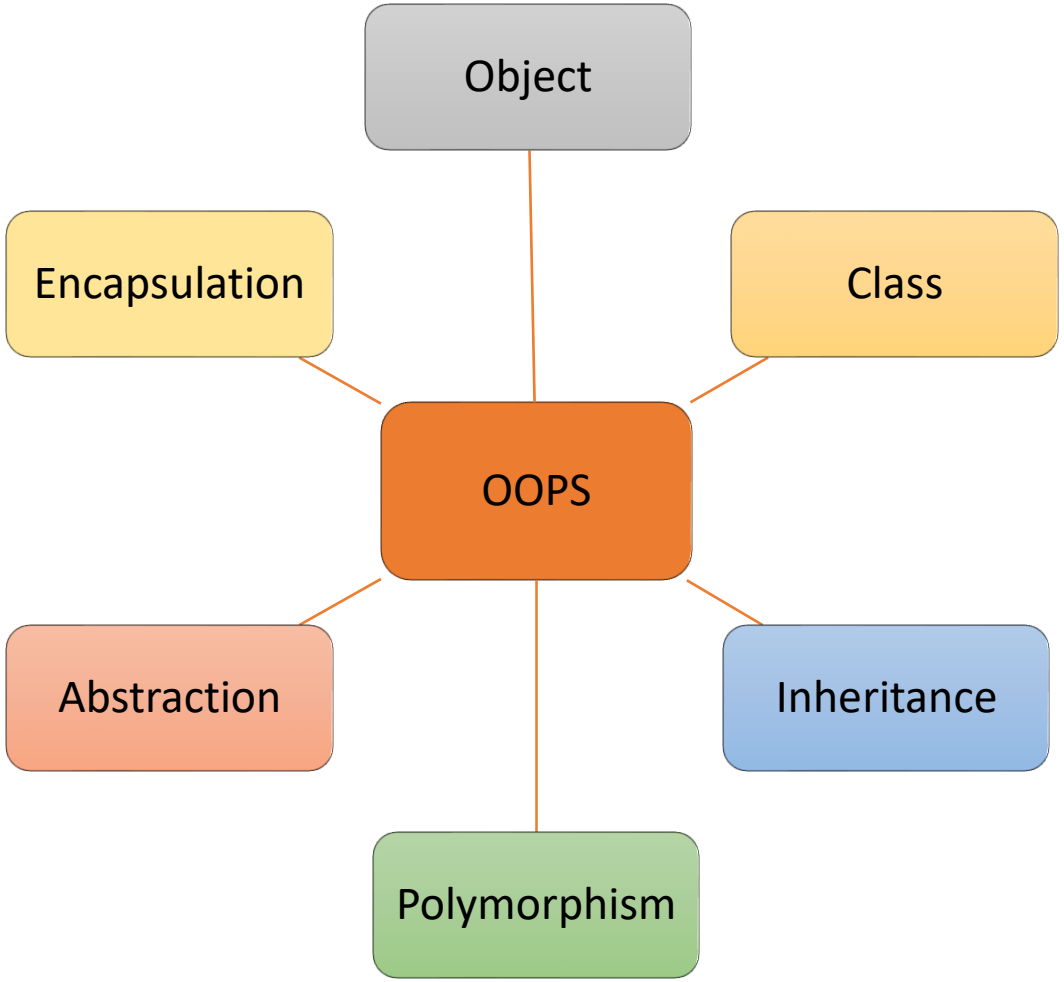
OOPs

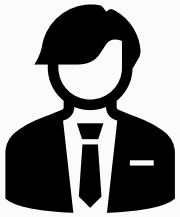
OOP stands for **Object-Oriented Programming**, which means OOPS is a way to create software around **objects**.

OOPs provide a **clear structure** for the software's and web applications.



- A class is a **blueprint/ template** for creating objects.
- An object is an **instance** (real world entity) of a class.

Employee Management System (abc.com)



When a new employee joined

Employee class (template)
- Fields: exp, name
- Functions: calculateSalary()

Object of Employee class(emp)
- Fields: emp.exp = 15, emp.name = Happy
- Functions: emp.calculateSalary()

back to chapter index

❖ Steps to implement class & object:

1. Create a class

2. Define class members inside it

3. Create object of the class at client(main())

4. Call the class member using the object created

```java
public class Employee {  // Class

  private int exp; // 1. Field

  public Employee() {} // 2. Constructor

  public double calculateSalary() { // 3. Method
    int salary = exp * 50000;
    return salary;
  }

  public static void main(String[] args) {
    // Create an Employee(emp) object
    Employee emp = new Employee();

    emp.exp = 5;

    double salary = emp.calculateSalary();

    System.out.println(salary);
    //Output: 250000
  }
}
```

back to chapter index

❖ **Class members are:**

1. Field	A field is a variable of any type. It is basically holds the data.
2. Constructor	A constructor is a method in the class which gets executed when a class object is created.
3. Method	A method is a block of code that performs a specific task.

```java
public class Employee {  // Class

  private int exp; // 1. Field

  public Employee() {} // 2. Constructor

  public double calculateSalary() { // 3. Method
    int salary = exp * 50000;
    return salary;
  }

  public static void main(String[] args) {
    // Create an Employee(emp) object
    Employee emp = new Employee();

    emp.exp = 5;

    double salary = emp.calculateSalary();

    System.out.println(salary);
    //Output: 250000
  }
}
```

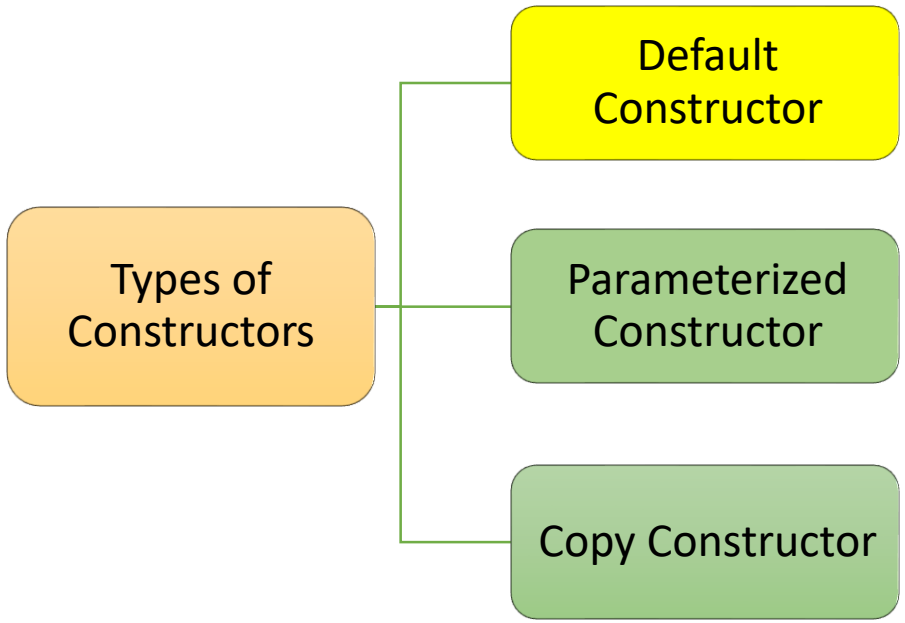
Constructor

A constructor is a special type of method that is called when the object of class is created.

A constructor has the same name as the class and does not have a return type, not even void.

```
public class Car {

  public Car() {
    System.out.println("my car");
  }

  public static void main(String[] args) {

    Car car = new Car();

  }
  // output: my car
}
```

- A default constructor in Java is a constructor which is **automatically provided by Java** if no other constructors are explicitly defined.



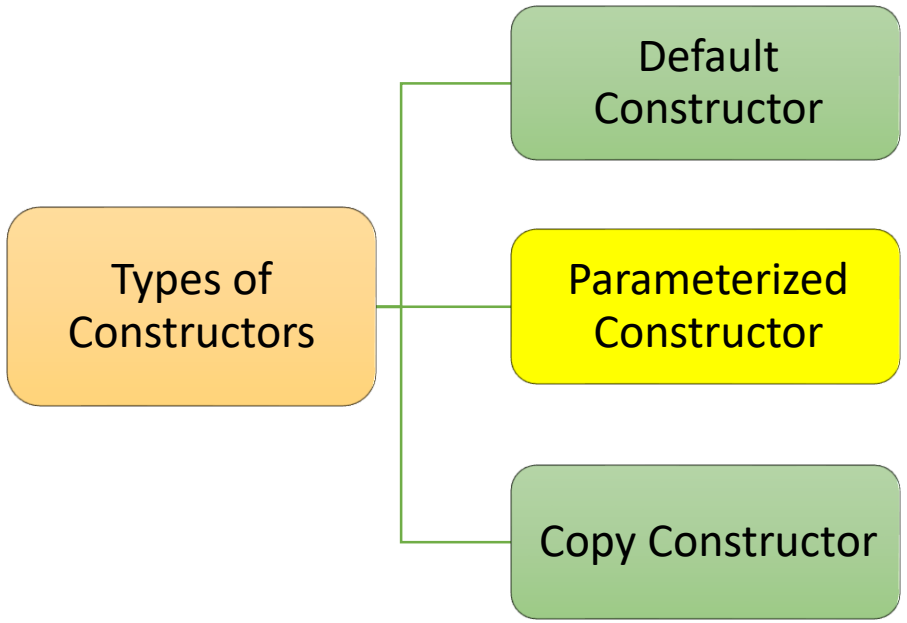

```java
public class Car {

 }
```

JVM (JIT Compiler)

```java
public class Car {

    // Default Constructor
    public Car() { }

 }
```

back to chapter index

- **Definition:** A parameterized constructor is a constructor **with parameters**.
- **Use:** It is used to initialize the specific field values of the class provided during object creation.




```java
public class Employee {

  private String name;

  // Parameterized constructor
  public Employee(String name) {
    this.name = name; // field initialization
  }

  public String getName() {
    return name;
  }

  public static void main(String[] args) {

    Employee emp = new Employee("Happy");

    System.out.println(emp.getName());
  }
}
```

back to chapter index

- **Definition:** Constructor overloading is the concept of having **multiple constructors** within the same class, each with a different parameter list.
- **Use:** Constructor overloading **enhances the usability of classes** by providing multiple ways to initialize objects based on varying requirements and input parameters.

```java
public class Rectangle {
  private int width;
  private int height;

  public Rectangle(int width, int height) {
    this.width = width;
    this.height = height;
  }

  public Rectangle(int side) {
    this.width = side;
    this.height = side;
  }

  public static void main(String[] args) {

    Rectangle rec = new Rectangle(10, 20);
    System.out.println(rec.width * rec.height); // 200

    Rectangle squ = new Rectangle(5);
    System.out.println(squ.width * squ.height); // 25
  }
}
```

back to chapter index

- Constructor chaining is the process of one constructor calling another constructor from the same class.

```java
public class Vehicle {

  private String brand;

  public Vehicle(String brand) {
    this.brand = brand;
  }

  public Vehicle() {
    // Constructor chaining (Calls the other constructor)
    this("Honda");
  }

  public String getBrand() {
    return brand;
  }

  public static void main(String[] args) {
    Vehicle vehicle = new Vehicle();
    System.out.println(vehicle.getBrand()); // Honda
  }
}
```

back to chapter index

- A copy constructor is used to create a new object as a copy of an existing object.
- A copy constructor typically accepts an instance of the same class as a parameter.

```java
public class Student {

  private String name;

  // Parameterized Constructor
  public Student(String name) {
    this.name = name;
  }

  // Copy constructor
  public Student(Student other) {
     // Copy name from the other object
     this.name = other.name;
  }

  // Getter method for name
  public String getName() {
    return name;
  }
```

```java
public static void main(String[] args) {
    Student stu1 = new Student("Happy");

    // Create object stu2 as a copy of stu1 object
    Student stu2 = new Student(stu1);

    System.out.println(stu1.getName()); //Happy
    System.out.println(stu2.getName()); //Happy
  }
}
```

back to chapter index

- No, constructors do not have a return type, not even void.

```
public class Employee {

  // Comiple time error
  public void Employee() {

    System.out.println("Happy");

  }
}
```

back to chapter index

- If no constructor is explicitly defined inside a class, the Java compiler automatically provides a default constructor for that class.

```
public class Employee
{

 }
```

JVM

```
public class Employee {
    // Default Constructor
    public Employee() {

    }
 }
```

- Super() keyword is used to call the constructor of superclass from subclass constructor, but it is optional to write.

```
public class Superclass {

  public Superclass() {
    System.out.println("from superclass");
  }
}
```

```
public class Subclass extends Superclass {

  public Subclass() {
    super(); // (optional - automatically placed)
    System.out.println("from subclass");
  }

  public static void main(String[] args) {
    // Creating an object of Subclass
    Subclass subclass = new Subclass();
  }
}
// Output: from superclass
// from subclass
```

back to chapter index

- Access specifiers (access modifiers) are keywords used to set the **access level** for classes, variables(fields), methods, and constructors.

Access Specifier	Within Class	Within Package	Subclass (Same Package)	Subclass (Different Package)	Outside Package
public	Yes	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	Yes	No
Default (no specifier)	Yes	Yes	Yes	No	No
private	Yes	No	No	No	No

```java
public class Student {
  public String name = "Happy";
  private String city = "Delhi";
}
```

```java
public class ScienceStudent {

  public void GetStudent() {
    Student student = new Student();
    System.out.println(student.name);

    // Error: private not accessible
    System.out.println(student.city);
  }
}
```

- The default access specifier(package-private access) is used when no explicit access specifier is provided. It controls the accessibility of member within the same package.

Access Specifier	Within Class	Within Package	Subclass (Same Package)	Subclass (Different Package)	Outside Package
public	Yes	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	Yes	No
Default (no specifier)	Yes	Yes	Yes	No	No
private	Yes	No	No	No	No

```java
package AccessSpecifiers;

public class Student {

  public String name = "Happy";

  private String city = "Delhi";

  int age = 39;
}
```

Same package

```java
package AccessSpecifiers;
public class ScienceStudent {

  public void GetStudent() {
    Student student = new Student();

    System.out.println(student.name); // public: no error
    System.out.println(student.city); // private: error
    System.out.println(student.age); // default: no error
  }
}
```

Different package

```java
package AccessSpecifiers1;
import AccessSpecifiers.Student;
public class MathsStudent {

  public void GetStudent() {
    Student student = new Student();

    System.out.println(student.name); // public: no error
    System.out.println(student.city); // private: error
    System.out.println(student.age); // default: error
  }
}
```

back to chapter index

- this keyword refers to the current instance of the class.
- Use of this keywords: Differentiate between instance variables and parameters with the same name in setter methods.

```java
public class Employee {

  private int exp;

  public void setExp(int exp) {

    this.exp = exp; // this.name is class field
  }

  public static void main(String[] args) {
    Employee emp = new Employee();

    //emp.exp = 10; // not recommended
    emp.setExp(10);
    System.out.println(emp.exp);
  }
}
```

- Using the same names for class fields and parameter names in setter methods can lead to cleaner, more readable code

```java
public class Employee {

  private int exp;

  public void setExp(int exp) {

    this.exp = exp; // this.name is class field
  }

  public static void main(String[] args) {
    Employee emp = new Employee();

    //emp.exp = 10; // not recommended
    emp.setExp(10);
    System.out.println(emp.exp);
  }
}
```

- Getter methods are used to **retrieve the values** of private fields of a class.
- Setter methods are used to **modify or set the value** of a private field of a class.

```java
public class Employee {

  private int exp; // Field

  public int getExp() { // Getter method
    return exp;
  }

  public void setExp(int exp) { // Setter method
    this.exp = exp;
  }

  public static void main(String[] args) {
    Employee emp = new Employee();

    //emp.exp = 5; // Not recommended
    emp.setExp(5); // Set exp using setter method

    System.out.println(emp.getExp());
    // Output: 5
  }
}
```

back to chapter index

❖ **Advantages of getter and setter methods:**

1. **Data Validation:** Setter methods can include validation logic to ensure that the assigned values are valid.
2. **Data Access:** Getter methods can control how a field is accessed.
3. **Encapsulation/ Abstraction:** Getter and setter methods **hide the internal implementation** details of a class and expose only necessary information.

```java
public class Employee {
  private int exp; // Field

  public int getExp() { // Getter method
    return exp * 2;
  }

  public void setExp(int exp) { // Setter method
    if (exp < 0) { // Validate experience
      System.out.println("Incorrect Experience");
      return;
    }
    this.exp = exp;
  }

  public static void main(String[] args) {
    Employee emp = new Employee();

    //emp.exp = 5; // Not recommended
    emp.setExp(5); // Set exp using setter method

    System.out.println(emp.getExp()); // Output: 10
  }
}
```

back to chapter index