Core java Notes

Datatypes :

- 2 types :
    1. Primitive
    2. Non-primitive
- **Primitive** :
    1. Int
    2. Char
    3. Float
    4. Long
    5. Double
    6. Boolean
    7. Byte
    8. Short
- **Non-primitive :**
    1. String
    2. Array
    3. Class
    4. Object, etc.

- **Access Modifiers :** Access modifiers help to restrict the scope of the class, constructors, variables, methods or data members.

- **Types of access modifiers :-**

1. **Default**- When no access modifier is specified for a class, method, or data member – It is said to be having the default access modifier by default. The data members, classes, or methods that are not declared using any access modifiers i.e. having **default access modifiers are accessible only within**

the same package.

2. **Private**: A class, method or data members can be declared private using private.
   The data member or methods declared private can only be accessible within the class in which they are declared.
   Any other class of the same package will not be able to access these members.
   Top-level classes or interfaces can not be declared as private because
   ➔ private means “only visible within the enclosing class”.
   ➔ protected means “only visible within the enclosing class and any subclasses”

3. **Protected**: The data members or methods that are declared protected can only be accessible within the same package or subclasses in the different package.

4. **Public**: The data members or methods declared as public are accessible from anywhere in the program. There is no restriction on the scope of public data members.

	Default	Private	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package Subclass	Yes	No	Yes	Yes
Same Package Non-Subclass	Yes	No	Yes	Yes
Different Package Subclass	No	No	Yes	Yes
Different Package N0on-Subclass	No	No	No	Yes

Object Oriented Programming with java

- OOPs is a programming paradigm/methodology.
- Paradigm
    - Object oriented programming paradigm
    - Procedural paradigm
    - Functional paradigm
    - Logical paradigm
    - Structural paradigm
- 6 main pillars of OOPs:
    - Class
    - Objects & methods
    - Inheritance
    - Polymorphism
    - Abstraction
    - Encapsulation

- **Class:-**
    - Class is the collection of objects.
    - Class is not a real-world entity it is just a template or blueprint or prototype.
    - Class does not occupy memory.
    - Syntax:
        - Access modifier class ClassName
          {
            - Methods
            - Constructors
            - Field
            - Objects
            - Nested class

          }

- **Methods:**
    - A set of codes which performs a particular task.
    - Advantages:
        - Code reusability
        - Code optimization
    - Syntax:-
        - Access-modifier return-type methodeName(list of parameters)
          {

          }

- **Object**
    - Object is an instance of a class.
    - Object is a real world entity.
    - Object occupies memory.
    - Object consist of:-
        - Identity: name
        - State/Attribute: color, bread, age, etc(in case of dog)
        - Behavior: eat, run.
    - Ways to create an object:
        - new keyword
        - newInstance() method
        - clone() method
        - Deserialization
        - Factory methods
    - We can create object by three steps:
        - Declaration: Animal buzo;
        - Instantiation: buzo = new Animal();
        - Initialization: Animal buzo = new Animal();

    - **Initializing object:**
        - There are three ways to initialize object:-
            1. By reference variable

```java
public class AIML {

    String name;
        String enroll;
        int age;
        String depart = "CSE-AIML";
    Run | Debug
    public static void main(String[] args) {
        AIML student1 = new AIML();
        student1.name = "Mukesh";
        student1.enroll = "0827AL201033";
        student1.age = 21;

        System.out.println(student1.name + ", " +
        student1.enroll + ", " + student1.age + ", " + student1.depart);
    }
}
```

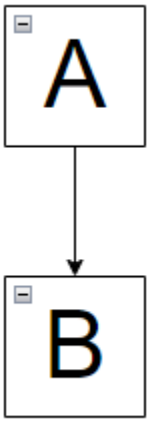
2. By constructors:-
    - **Constructors -**
        - Constructor is a special type of method used to initialize objects in java.
        - It has the same name as the class name.
        - It does not have any return type, not even void.
    - Use of constructor-
        - To initialize an object without using a reference variable.
        - With the use of constructor, we don't have to write extra line of code like,
            - obj.name = "abcd";
            - obj.id = 101;
        - Constructors initializes every reference variable in an object with a default value, as soon as the object is created.

```java
//intializing object by contructor
public class MyClass {
    String name;
    int emp_id;

    public MyClass(String name, int emp_id){
        this.name = name;
        this.emp_id = emp_id;
    }
    public static void main(String args[]) {
      MyClass obj1 = new MyClass("abc", 101);
      MyClass obj2 = new MyClass("xyz", 102);

      System.out.println(obj1.name);
      System.out.println(obj1.emp_id);

      System.out.println("");

      System.out.println(obj2.name);
      System.out.println(obj2.emp_id);

    }
}
```

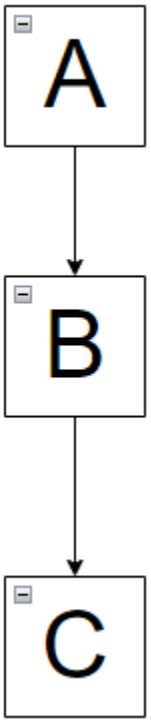
- **Inheritance**
    - Inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object.
    - IS-A-relationship.
    - The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
    - Use of inheritance
        - For Method Overriding (so runtime polymorphism can be achieved).
        - For code reusability.
        - Syntax of inheritance in java.
            - Class child extends parent {

              }

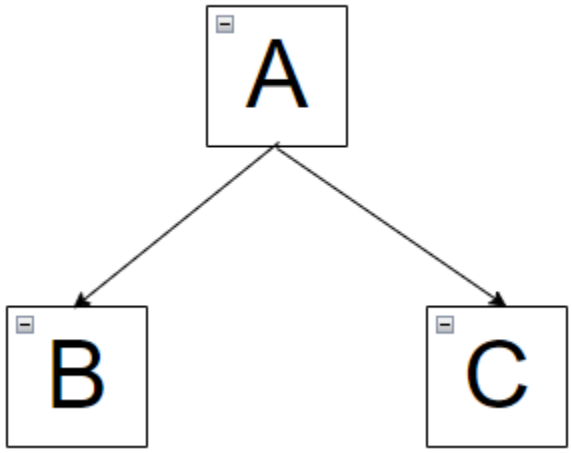
- Disadvantage of inheritance:
    - Classes are tightly coupled(change in parent class will lead to change in every child class).
- **Types of inheritance :-**
    - Single inheritance:



    - Multilevel inheritance

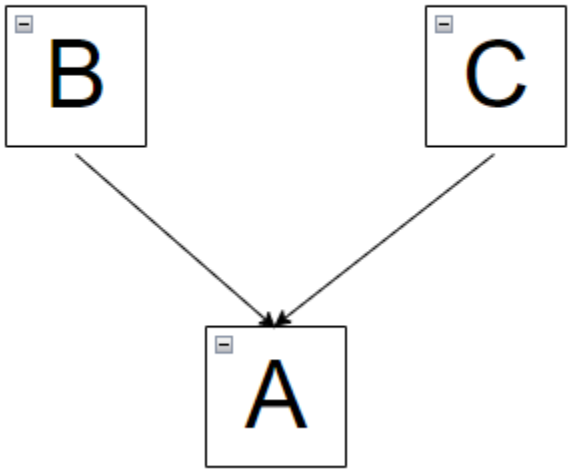


- Hierarchical inheritance



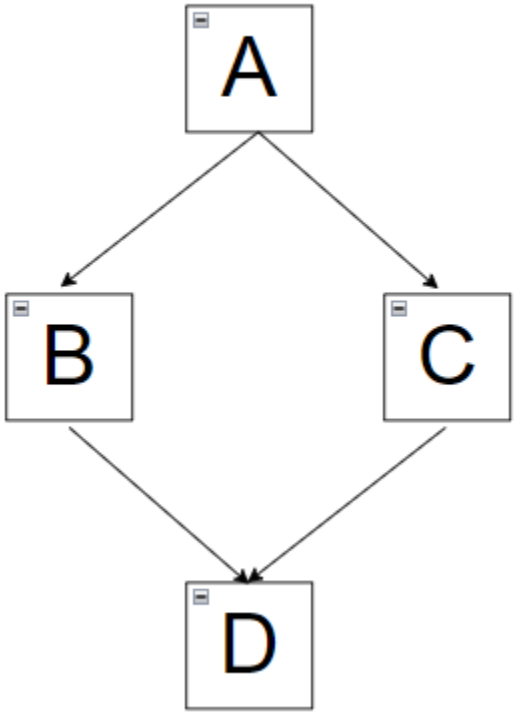
-

- Multiple inheritance



-

- Hybrid inheritance



-

- Java only supports single, multilevel and hierarchical inheritance.
- Multiple and hybrid inheritance causes ambiguity error.
- We cannot inherit **all** the properties of the parent class in a child class.
- Some properties cannot be inherited like constructor and private methods.
- Private methods cannot be inherited in a child class.

- **Types of relationships between classes in java**
  - Inheritance (IS-A relationship)
  - Association (HAS-A relationship)

    - Two types:-
        - Aggregation (Weak bonding)
        - Composition (Strong bonding)
- Advantages of relationship between classes:-
    - Code reusability
    - Cost cutting
    - Reduce redundancy (Removing useless code)
- **Association**
    - Association is a type of relationship between two or more classes, where objects of one class are connected to objects of another class.
    - It is a HAS-A type of relationship.
    - Types of association:-
        1. One-to-One (1:1) Association:
            - This is a relationship where each object of one class is associated with exactly one object of another class.
            - Example: A **'Person'** class may have a one-to-one association with an **'Address'** class, where each person has exactly one address.

        2. One-to-Many (1:N) Association:
            - In a one-to-many association, each object of one class is associated with multiple objects of another class.
            - Example: A Department class may have a one-to-many association with an Employee class, where one department has multiple employees.

        3. Many-to-One (N:1) Association:
            - This is the reverse of one-to-many association. Many objects of one class are associated with exactly one object of another class.
            - Example: Multiple Students may be associated with one School in a many-to-one relationship.

4. Many-to-Many (N:N) Association:
    - In a many-to-many association, objects of one class are associated with multiple objects of another class, and vice versa.
    - Example: A Course class may have a many-to-many association with a Student class, where a student can enroll in multiple courses, and a course can have multiple students.

- In association classes are not tightly coupled change in one class will not affect the other.

```
public class Department {
    private String name;
    private List<Employee> employees;

    // Constructor, getters, setters, etc.
}

public class Employee {
    private String name;
    private String employeeId;

    // Constructor, getters, setters, etc.
}
```

-

- Forms of association
    - **Aggregation (Weak bonding)**
        - Aggregation in Java is a type of association between two classes that represents a "whole-part" relationship. It is a more specialized form of association where one class is considered as a container or holder for another class, but the contained class can exist independently.

Aggregation implies a relationship where the child can exist independently of the parent.

- For example ; A car can or cannot have a music player but a music player must have a car associated with it.

- **Composition (Strong bonding)**
  - Composition in Java is a strong form of association where one class contains an object of another class as a part, and the contained object has no meaning or purpose outside the context of the container class. In other words, the composed class is an integral part of the containing class, and it typically cannot exist or have a meaningful existence without the container.
  - For example : A car must have an engine and visa-versa.
  -

```
public class Car {
    private Engine engine;
    private Wheel[] wheels;

    public Car() {
        this.engine = new Engine();
        this.wheels = new Wheel[4];
        // Initialize wheels and perform other setup
    }

    // Other methods and functionality of the Car class
}

public class Engine {
    // Engine-related attributes and methods
}

public class Wheel {
    // Wheel-related attributes and methods
}
```

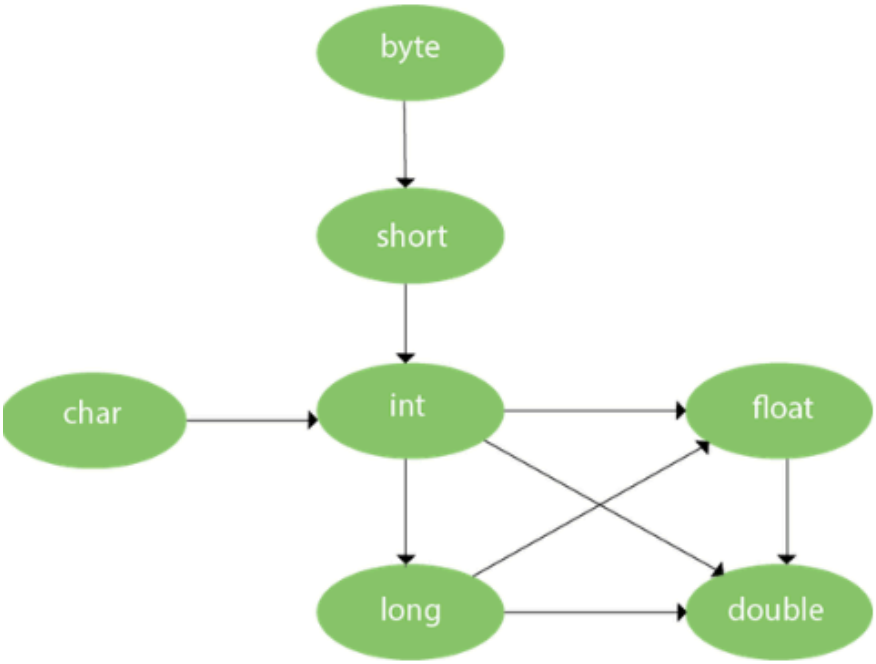
- **Polymorphism**
    - ‘Poly’ means many and ‘morphs’ means form.
    - Polymorphism means having different forms.
    - Polymorphism is a concept by which we can perform a single action in different ways.
    - Polymorphism is a concept which refers that, a variable, method or object that can have more than one form.
    - There are two types of polymorphism:-
        - Compile time polymorphism
        - Runtime polymorphism
    - **Compile time polymorphism-**
        - Also known as static polymorphism.
        - This type of polymorphism can be achieved by method overloading or operator overloading
        - Java does not support operator overloading.
    - **Method overloading:-**
        - When there are more than one methods with the same name but different parameters then it is known aws method overloading.
        - There are two ways to overload a method:-
            - By changing the no of parameters.
            - By changing the data type of parameters.
        - Method overloading :
            - Same name
            - Same class
            - Different parameters
                - No of parameters
                - Type of parameters
                - Sequence of parameters
            - **Method signature**
                - Method signature is used by the compiler to differentiate the methods.
                - It consists of three parts:
                    - Name of method
                    - Number of methods
                    - Types of parameters

- We can declare one method as static and one method as non-static
- **Is it possible to have two methods in a class with the same method signature but different return types?**
  Ans: No, the compiler will give a duplicate method error. Compiler checks only method signatures for duplication, not the return types. If two methods have the same method signature, straight away it gives a compile time error.
- **Can we overload main method?**
  Ans: Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls the main() method which receives string arrays as arguments only.
- Different cases of method overloading
- **Case 1:**

```
public class Test {
    void show(int a){
        System.out.println(a);
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.show('a');
    }
}
```

-
  - Output: 65(ASCII value of ‘a’)

- **Automatic type promotion**
  One type can be promoted to another implicitly if no matching datatype is found. Below is the diagram:



-

- **Case 2:**

```
import java.util.*;
public class Test {
    void show(Object a){
        System.out.println(a);
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.show('a');
    }
}
```

- Output : a

```
import java.util.*;
public class Test {
    void show(Object a){
        System.out.println(a);
        System.out.println("Object method");
    }
    void show(String s){
        System.out.println(s);
        System.out.println("String method");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.show('a');
        t.show("abc");
        t.show(null);
   }
}
```

-

Output :
a
Object method
abc
String method
null
String method

- While resolving Overloaded methods, Compiler will always give precedence to child type argument then compared with parent type argument.

- **Case 3:**

```
import java.util.*;
public class Test {
    void show(StringBuffer a){
        System.out.println("StringBuffer method");
    }
    void show(String s){
        System.out.println("String method");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.show("abc");
        t.show(new StringBuffer("abc"));
        // t.show(null); Ambuguity
   }
}
```

- Output:
  String method
  StringBuffer method
- **Case 4:**

```
import java.util.*;
public class Test {
    void show(int a, float b){
        System.out.println("int float method");
    }
    void show(float a, int b){
        System.out.println("float int method");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.show(10, 20.3f);
        t.show(20.3f, 10);
        // t.show(10, 20); Ambuguity
   }
}
```

  Output:
  int float method
  float int method

- **Case 5:**

```
import java.util.*;
public class Test {
    void show(int a){
        System.out.println("int method");
    }
    void show(int... a){
        System.out.println("varargs method");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.show(10);
        t.show(10, 20, 30);
        t.show();
    }
}
```

Output:
int method
varargs method
varargs method

- varargs method has the least priority.

- **Runtime polymorphism**
  - Dynamic polymorphism
  - It is a process in which an overridden method is resolved during runtime rather than at compile time.
  - Can be achieved by method overriding
  - **Method overriding:**
    - Same name
    - Different class
    - Same argument(parameter):
      - No of parameter
      - Type of parameter
      - Sequence of parameter
    - Inheritance

```
class Test {
    void show(){
        System.out.println("Test class");
    }
}
class Xyz extends Test{
    void show(){
        System.out.println("Xyz class");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.show();

        Xyz x = new Xyz();
        x.show();
    }
}
```

-
- Output:
  Test class
  Xyz class
- Method overriding allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclass or parent class. The implementation in the subclass overrides (replaces) the implementation in the superclass by providing a method that has the same name, same parameters or signature, and same return type as the method in the parent class.

- **Do overriding methods have the same return type(or subtype) ?**
  Ans: From Java 5.0 onwards it is possible to have different return types for an overriding method in child class, but the child's return type should be sub-type of parent's return type. This phenomenon is known as **covariant return type**

```
class Test {
    Object show(){
        System.out.println("Test class");
        return 0;
    }
}
class Xyz extends Test{
    String show(){
        System.out.println("Xyz class");
        return null;
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.show();

        Xyz x = new Xyz();
        x.show();
    }
}
```

- **Overriding and Access-modifiers**
  Ans: The access modifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protracted instance method in the super-class can be made public, but not private, in the subclass. Doing so will generate compile-time errors
- **Error:**

```
class Test {
    public void show(){
        System.out.println("Test class");
    }
}
class Xyz extends Test{
    void show(){
        System.out.println("Xyz class");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.show();

        Xyz x = new Xyz();
        x.show();
    }
}
```

- **Correct code:**

```
class Test {
    void show(){
        System.out.println("Test class");
    }
}
class Xyz extends Test{
    public void show(){
        System.out.println("Xyz class");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.show();

        Xyz x = new Xyz();
        x.show();
    }
}
```

- **Overriding and exception-handling:**
- Below are two rules to note when overriding methods related to exception handling.
    - **Rule 1:**
      If the superclass overridden method does not throw an exception, a subclass overriding method can only throw the unchecked exception, throwing checked exception will lead to compile-time error.
    - **Rule 2:**
      If the superclass overridden method does throw an exception, a subclass overriding method can only throw the same, subclass exception, Throwing parent exception in exception hierarchy will lead to compile time error. Also there is no issue if the subclass overridden method is not throwing any exception.
- **Overriding and abstract method**
    - Abstract methods in an interface or abstract classes are meant to be overridden in derived concrete classes otherwise compile time error will be thrown.
    - Invoking the overridden method from subclass.

        - We can call the parent class method in the overriding method using the super keyword.
    - **Important points:**
        - Final methods can not be overridden.
        - Static methods can not be overridden; when you define a static method with the same signature as a static method in base class, it is known as **method hiding.**
        - Private methods can not be overridden; private methods cannot be overridden as they are bonded during compile time. Therefore we can't override private methods in a subclass.
- **Overriding and synchronized/strictfp method**
    - The presence of synchronized/strictfp modifiers with methods have no effect on the rules of overriding, i.e. it's possible that a synchronized/strictfp method can override a non synchronized/strictfp one and vise-versa.

Abstraction

- Abstraction is hiding the internal implementation and just highlighting the setup services that we are offering.
- Abstraction is hiding the implementation and showing only functionality to the user.
-

Abstraction	Encapsulation
- Abstraction means detail hiding.(Implementation hiding) - Data abstraction deals with exposing the interface to the user and hiding the details of implementation.	- Encapsulation is data hiding (Information hiding) - Encapsulation groups together data and methods that act upon the data.

- There are two ways to achieve abstraction in java

1. By using abstract class. (0 to 100%)
2. By using Interface (100%)

1. **Abstract class**
    - **Abstract method:** A method without implementation is known as abstract method.
    - A abstract method must always be declared in an abstract class or we can say that if a class contains an abstract method then the class must be declared as abstract.
      But it is not mandatory for an abstract class to have an abstract method. An abstract class can have abstract methods as well as concrete methods.
    - If a regular class extends an abstract class then the class must have to implement all the abstract methods of the parent abstract class or it has to be declared abstract as well.
    - Abstract methods is an abstract class are meant to be overridden in the derived concrete classes otherwise compile time error will be thrown.
    - Abstract classes can not be instantiated; means we can't create object of an abstract class.

```java
abstract class Vehicle{
    abstract void start();
}
Comment Code
class Car extends Vehicle{
    void start(){
        System.out.println(x:"Car starts with a key.");
    }
}
Comment Code
class Scooter extends Vehicle{
    void start(){
        System.out.println(x:"Scooter starts with a kick");
    }
    Run | Debug
    public static void main(String[] args) {
        // Vehicle v = new Vehicle();

        Car c = new Car();
        c.start();

        Scooter s = new Scooter();
        s.start();;
    }
}
```

-

Interface:-

- Interfaces are same as abstract class but having all the methods of abstract type.
- All methods in an interface are abstract methods.
- Interfaces are the blueprint of the class. It specifies what a class must do and not how.
- It is used to achieve abstraction.
- It supports multiple inheritance .
- It can be used to achieve loose coupling.
- Syntax:

```
Interface InterfaceName{
        methods //only abstract public methods
        Fields  //public static final
}
```

- In java 8 and 9 version we can also add concrete methods in interfaces but the condition is that the methods should be declared default.

  After Java 8:

  default void display(){

  }

- We can also create **static** methods in an interface in java 8.
- After java 9 we can also create **private** methods in an interface.

```
interface I1{
    void show();
    public abstract void display();
    //After java 8
    default void m1(){
        System.out.println(x:"Default method can be added");
    }
    static void m2(){
        System.out.println(x:"Static methods can be added");
    }
    //After java 9
    private void m3(){
        System.out.println(x:"Prvate methods can be added");
    }
}
class InterfaceExample implements I1{
    public void show(){
        System.out.println(x:"Interface display");
    }
    Run | Debug
    public static void main(String[] args){
        InterfaceExample in = new InterfaceExample();
        in.show();
    }
}
```

-

- By using interface we can achieve **Multiple Inheritance**
  (I1 ➡️ class ⬅️ I2)

```java
interface I1{
    void show();
}
interface I2{
    void display();
}
class InterfaceExample implements I1,I2{
    public void show(){
        System.out.println(x:"Interface 1");
    }
    public void display(){
        System.out.println(x:"Interface 2");
    }
    Run | Debug
    public static void main(String[] args){
        InterfaceExample in = new InterfaceExample();
        in.show();
        in.display();
    }
}
```

-

Encapsulation:

- Encapsulation in java is a mechanism of wrapping the data (variables) and code acting upon that data (methods) together as a single unit.
- There are two steps to achieve encapsulation:
    1. Declare the variables of the class as private.
    2. Provide public getter methods and setter methods to modify.
- In encapsulation, the variables of a class will be hidden from the other classes and can accessed only through the methods of their current class. This concept is known as **data hiding.**
- In simple words, encapsulation means declaring the variables of a class as private and accessing them through getter and setter methods.

```java
class Employee {
    private int emp_id; //Data hiding

    public void setEmpId(int emp_id1){
        emp_id = emp_id1;
    }

    public int getEmId(){
        return emp_id;
    }
}
```

//Encapsulation

```java
class Company {
    Run | Debug
    public static void main(String[] args) {
        Employee e = new Employee();
        e.setEmpId(emp_id1:101);
        System.out.println(e.getEmId());
    }
}
```

Static Keyword:

-

Access Modifiers	Non-access modifiers
- Public - Private - Protected - Default (No modifier)	- static - final - abstract - synchronized - transient - volatile - strictfp

- Non-access modifiers are keywords in Java that provide additional information about the characteristics of a class, method, or variable to the JVM. They are used to introduce functionalities such as:
    - final: - indicates that the variable cannot be initialized twice.
    - static: - used for creating class methods and variables.
    - abstract: - used for creating abstract classes and methods.

- synchronized: - used for threads.
- volatile: - used for threads.

These modifiers are not related to the level of access, but they provide special functionality when specified.

- Static keyword **can be** used in following cases:
  - With a class variable
  - With methods
  - With blocks
  - With inner class (Nested class)
- Static keyword **cannot be** used in following cases:
  - With local variable
  - With outer class in nested classes

```
class Test {
    static int a = 10;
    void something(){
        static int b = 20; //error
    }
}
```

-

- Static variables belongs to the class not object; it means we can access the static variable directly from the class name, there is no need to create the object of the class in order to access the static variable

```
class Test {
    int a = 10;
}
class demo{
    Run | Debug
    public static void main(String[] args) {
        System.out.println(Test.a);
        /*Cannot make a static reference
        to the non-static field Test.a*/
    }
}
```

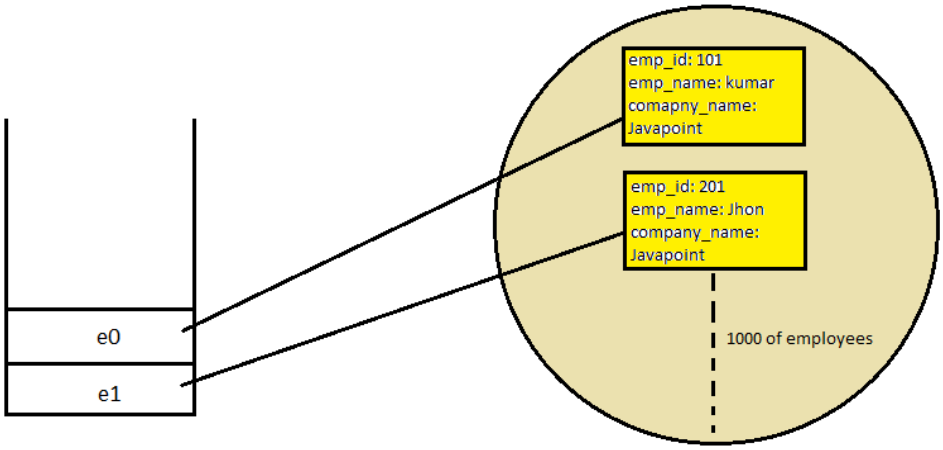
-

```
class Test {
    static int a = 10;
}
class demo{
    Run | Debug
    public static void main(String[] args) {
        System.out.println(Test.a);
        //Succesfully compiled
    }
}
```

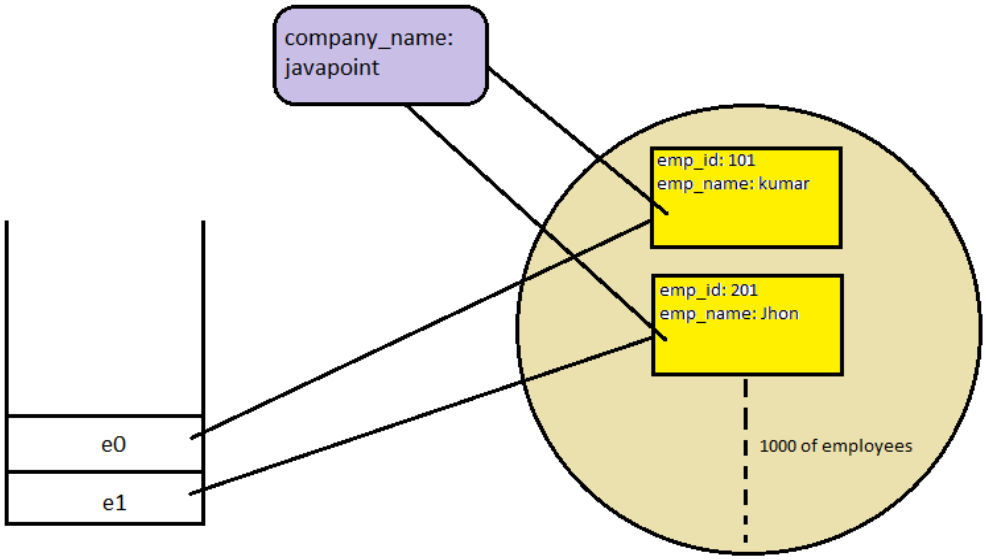
-

- Significance of **static** keyword:
    - Static keyword is used for memory management:

```
public class Employee {
    int emp_id;
    String emp_name;
    String company_name;

    public Employee(int emp_id, String emp_name, String company_name) {
        this.emp_id = emp_id;
        this.emp_name = emp_name;
        this.company_name = company_name;
    }

    void display() {
        System.out.println(emp_id + " " + emp_name + " " + company_name);
    }

    Run | Debug
    public static void main(String[] args) {
        Employee e0 = new Employee(emp_id:101, emp_name:"kumar", company_name:"Javapoint");
        Employee e1 = new Employee(emp_id:201, emp_name:"Jhon", company_name:"Javapoint");
        Employee e2 = new Employee(emp_id:301, emp_name:"Chris", company_name:"Javapoint");

        e0.display();
        e1.display();
        e2.display();
    }

}
```



- In the above code we can see that as we are initializing the class variables through the constructor, **the company name** is the same for all the objects or employees. Still, we have to provide the company name in the constructor again and again. So every object contains a variable company name.
- This will take a lot of useless memory.
- By using static keyword we can save this memory.
- When a variable is declared static then a copy of the variable is created and shared among all the objects at class level. Thus company name "Javapoint" will be shared among objects e0, e1 and e2.
- By this way there will only two variables created for a object, unlike three company name there will be only one variable company name that will be shared among all the objects.



```java
public class Employee {
    int emp_id;
    String emp_name;
    static String company_name = "JavaPoint";

    public Employee(int emp_id, String emp_name) {
        this.emp_id = emp_id;
        this.emp_name = emp_name;
    }

    void display() {
        System.out.println(emp_id + " " + emp_name + " " + company_name);
    }

    Run | Debug
    public static void main(String[] args) {
        Employee e0 = new Employee(emp_id:101, emp_name:"kumar");
        Employee e1 = new Employee(emp_id:201, emp_name:"Jhon");
        Employee e2 = new Employee(emp_id:301, emp_name:"Chris");

        e0.display();
        e1.display();
        e2.display();
    }

}
```

-

- Static variable takes memory in class area or method area but object variables take memory in heap area.
- Static variables make our program memory efficient.
- Static variables can be used to refer to the common property or the value for all the objects.
  For example: company name of employees , college name of students, etc.
- Static variable only gets memory once at the time of class loading.
- Static variables can only be instantiated once.

```
class Test{
    static int count=0;
    public Test(){
        count++;
        System.out.println(count);
    }
    Run | Debug
    public static void main(String[] args) {
        Test t1 = new Test();
        Test t2 = new Test();
        Test t3 = new Test();
    }
}
```

-

Output:

1

2

3

- **Static Methods:**
  - Static methods belong to the class, not the object.
  - A static method can be accessed directly by the class name and doesn't need any object.
  - A static method can access only static data. It cannot access non-static data (instance data).

```
class Test{
    int i = 10;
    static int j = 20;
    static void display(){
        // System.out.println(i); error
        System.out.println(j);
    }
    Run | Debug
    public static void main(String[] args) {
        display();
    }
}
```

- A static method can call only static methods and cannot call non-static methods.

```
class Test{

    void methodOne(){
        System.out.println(x:"This is method one");
    }

    static void methodTwo(){
        System.out.println(x:"This is method two");
    }

    static void demo(){

        // methodOne();
        /*Cannot make a static reference to the
        non-static method methodOne() from the type Test*/

        methodTwo(); //succesfully compiled
    }

    Run | Debug
    public static void main(String[] args) {
        demo();
    }
}
```

- A static method cannot refer to “this” or “super” keyword in any way.

```
class Test{
    int i = 10;
    static void display(){
        // System.out.println(this.i);
        /*Cannot use this in static context */
    }
    Run | Debug
    public static void main(String[] args) {
        display();
    }
}
```

- Static methods are used for memory management, by declaring a method static we don’t have to create an object this save memory.
- Static methods can be called directly or with the class name.

```
class Test{
    static void show(){
        System.out.println(x:"Hello World");
    }
    Run | Debug
    public static void main(String[] args) {
        show();
        Demo.display();

    }
}
Comment Code
class Demo{
    static void display(){
        System.out.println(x:"Demo class method");
    }
}
```

-

Output:

```
Hello World
Demo class method
```

Static block:

- static
  {

  }

- Static block executed automatically when a class is loaded in the memory.
- Before java 1.6 version a static block can be executed without a main method, which means in the following program “Hello” can be printed.

```
class Test{
    static{
        System.out.println(x:"Hello");
    }
}
```

- But after the 1.6 version it is necessary to create the main method.

```
class Test{
    static{
        System.out.println(x:"Hello");
    }
    Run | Debug
    public static void main(String[] args) {

    }
}
```

Output:

```
Hello
```

- Sabse pehle static block hi execute hota h
- Multiple static blocks can be created in a class. (Execution will be from top to bottom).
- All static blocks are executed before the main method.
- **Use of static block:**
    - Static block executes at class loading hence at the time of class loading if we want to perform any activity, we have to define that inside the static block.
    - Code to load native methods are defined in static block.
    - Static block is used to initialize the static variables.

“this” keyword:

- “this” keyword is the reference variable that refers to the current object.
- “this” refers to the current class instance variable
  Without this keyword:-

```
class Xyz{
    int i;
    void setValue(int i){
        i = i; //both are local variables
    }
    void show(){
        System.out.println(i);
    }
}
Comment Code
class Test{
    Run | Debug
    public static void main(String[] args) {
        Xyz oj = new Xyz();
        oj.setValue(i:10);
        oj.show();
    }
}
```

Output: 0

With this keyword:-

```
class Xyz{
    int i;
    void setValue(int i){
        this.i = i;
    }
    void show(){
        System.out.println(i);
    }
}
Comment Code
class Test{
    Run | Debug
    public static void main(String[] args) {
        Xyz oj = new Xyz();
        oj.setValue(i:10);
        oj.show();
    }
}
```

Output: 10

- **Uses of “this” variable:**
    1. this keyword refers to the current class instance variable.
    2. This keyword can be used to invoke the current class method(implicitly).
    3. this() can be used to invoke the current class constructor.

```
class Test{
    Test(){
        System.out.println(x:"No argument constructor");
    }
    Test(int x){
        this();
        System.out.println(x:"Parametrized constructor");
    }
    Run | Debug
    public static void main(String[] args) {
        Test t = new Test(x:10);
    }
}
/*
Output:
No argument constructor
Parametrized constructor
*/
```

    4. this can be used to pass as an argument in the method call.

```
class Test{
    void m1(Test td){
        System.out.println(x:"This is m1 method");
    }
    void m2(){
        m1(this);
    }
    Run | Debug
    public static void main(String[] args) {
        Test t = new Test();
        t.m2();
    }
}
/*
Output:
This is m1 method
*/
```

5. this can be used to pass as an argument in the constructor call.

```
class ThisDemo{
    ThisDemo(Test td){
        System.out.println(x:"ThisDemo class constructor");
    }
}
Comment Code
class Test{
    void m1(){
        ThisDemo td = new ThisDemo(this);
    }
    Run | Debug
    public static void main(String[] args) {
        Test t = new Test();
        t.m1();
    }
}
/*
Output:
ThisDemo class constructor
 */
```

6. this can be used to return the current class instance from the method.

```
class Test{
    Test m1(){
        return this;
    }
    Run | Debug
    public static void main(String[] args) {
        Test t = new Test();
        t.m1();
    }
}
```

Super Keyword:

- “super” is a reference variable that refers to the immediate parent class object.

```
class superDemo{
    int a = 10;
}
Comment Code
class Test extends superDemo{
    int a = 20;
    void m1(int a){
        System.out.println(a); //30
        System.out.println(this.a); //20
        System.out.println(super.a); //10
    }
    Run | Debug
    public static void main(String[] args) {
        Test t = new Test();
        t.m1(a:30);
    }
}
```

- Uses of “super” keyword:
    - super keyword can be used to refer to the immediate parent class variable.

- super keyword can be used to invoke the immediate parent class method.

```
class superDemo{
    void show(){
        System.out.println(x:"superDemo class show method");
    }
}
Comment Code
class Test extends superDemo{
    void show(){
        super.show();
        System.out.println(x:"Test class show method");
    }
    Run | Debug
    public static void main(String[] args) {
        Test t = new Test();
        t.show();
    }
}
/*
Output:
superDemo class show method
Test class show method
 */
```

- super keyword can be used to invoke immediate parent class constructor.
  In case of constructor super() is used to refer to the immediate parent class constructor.
  super - keyword
  super() - use in case of constructor

```
class superDemo{
    superDemo(){
        System.out.println(x:"superDemo class constructor");
    }
}
Comment Code
class Test extends superDemo{
    Test(){
        super();
        System.out.println(x:"Test class constructor");
    }
    Run | Debug
    public static void main(String[] args) {
        Test t = new Test();
    }
}
/*
Output:
superDemo classs constructor
Test class constructor
 */
```

In the above example even if we don't write super() the output will be the same. Compiler automatically adds super().

```
class superDemo{
    superDemo(){
        System.out.println(x:"superDemo class constructor");
    }
}
Comment Code
class Test extends superDemo{
    Test(){
        System.out.println(x:"Test class constructor");
    }
    Run | Debug
    public static void main(String[] args) {
        Test t = new Test();
    }
}
/*
Output:
superDemo classs constructor
Test class constructor
 */
```

Final keyword:

- We can use final keyword in three cases:
  - ➔ **With variables:** If we create any final variable, it becomes constant, we cannot change the value of the final variable.

```
class Test{
    Run | Debug
    public static void main(String[] args) {
        int i = 10;
        i = i + 20;
        System.out.println(i); // 30

        final int j = 10;
        // j = j + 20; error
        System.out.println(j); // 10
    }
}
```

  - ➔ **With methods:** If we create a final method, it cannot be overridden.

```
class firstClass{
    final void m1(){
        System.out.println(x:"firstClass");
    }
}
Comment Code
class Test extends firstClass{
    void m1(){ //error
        System.out.println(x:"Test");
    }
    Run | Debug
    public static void main(String[] args) {

    }
}
```

➔ **With class:** If we create any final class, we cannot extend it or inherit it.

```
final class firstClass{

}
Comment Code
class Test extends firstClass{ //error
    Run | Debug
    public static void main(String[] args) {

    }
}
```

THE END