**OBJECT ORIENTED PROGRAMMING IN JAVA**

**Object Oriented programming** is a programming paradigm that revolves around the concept of objects. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

**BENEFITS OF OOPS:**

### 1. ****Reusability****

* Inheritance provides re-usability of code, additional features can be added to existing class without modifying it.
* **Example**: A Vehicle class can be inherited by Car and Bike classes, reusing common attributes like speed and color.

### 2. ****Scalability****

* **Class Hierarchies**: OOP supports building complex systems using class hierarchies, making it easier to manage and scale large software projects.
* **Example**: In a banking system, you can have a base class Account and derived classes like SavingsAccount and CheckingAccount.

### 3. ****Security****

* **Data Hiding**: By restricting direct access to the objects through encapsulation and access modifiers OOP helps in protecting the integrity of data.
* **Example**: Private member variables cannot be accessed directly from outside the class, reducing the risk of unintended data manipulation.

### 4. ****Real-World Modeling****

* **Object Representation**: OOP allows developers to create classes that model real-world entities, and instantiate objects that represent them, making the code more intuitive and aligned with how people think about the world.
* **Example**: Modeling a Car with attributes like engine, wheels, and behaviors like drive, brake.

### 5. ****Easier Troubleshooting****

* **Self-Contained Objects**: Since objects are self-contained, it is easier to debug issues within specific objects rather than a monolithic block of code.
* **Example**: If a bug is found in the Payment class, it can be fixed without necessarily affecting other parts of the system.

### 6. ****Better Collaboration****

* **Team Development**: Team members can work on different classes and objects without interfering with each other’s work.
* **Example**: One developer can work on the UserInterface class while another works on the Database class.

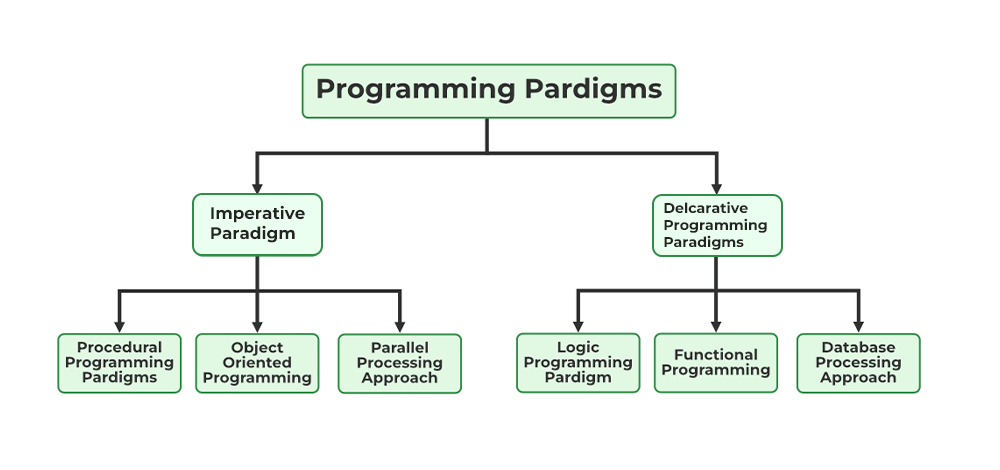
**OBJECT**

* An **object** is an instance of a class and represents a real-world entity.
* Takes up space in the memory and has an associated address.
* It is variable of the type class.
* When a class is defined, no memory is allocated, but memory is allocated when it is instantiated (i.e., an object is created).

**CLASS**

* A class is a blueprint or template for creating objects.
* It defines the data members and methods that objects of that class will contain.
* It is user defined data type that behave like built-in types of a programming language.

**PROGRAMMING PARADIGMS:** (methodology or philosophy of programming)



**1. Imperative Programming Paradigm:** Imperative programming tells the computer how to do the task by giving a set of instructions in a particular order. The order of execution of instruction needs to be explicitly defined by the programmer.

Procedural Programming Paradigm (top-down):

* The entire program is split into procedures (procedures aren’t functions).
* Procedure is a section of code that performs a specific task. Eg- for loop, the main purpose of for loop is to cause side effects and does not returns a value.
* Data and logic are two separate entities.
* Example: C program.

Object-Oriented Programming (OOP) (bottom-up):

* Entire program is built using classes and objects. Eg- JAVA

**2. Declarative Programming Paradigm:** Declarative programming focuses on what is to be executed rather than how it should be executed. Eg- SQL

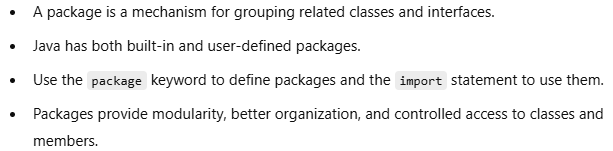
Logical Programming Paradigm:

* The program is not made up of instructions, rather it is made up of facts and clauses.
* The logic programming using its knowledge base tries to come up with a conclusion like true or false.
* Example: Prolog

Functional Programming Paradigm:

* The entire program is split into functions.
* Functions are first-class citizens, meaning they can be assigned to variables, passed as arguments, or returned from other functions.
* Each function receives some input as parameters and returns output
* Example languages: Haskell, Scala.

**Packages**

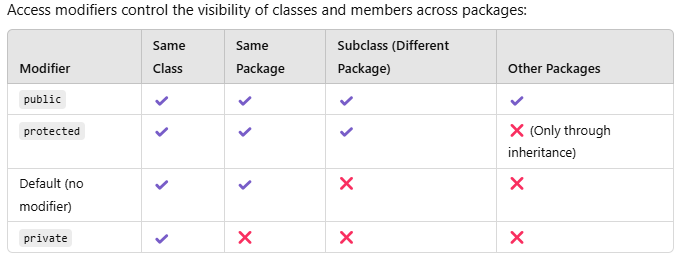


**ACCESS MODIFIERS:** can be used for class, member variables and member functions.

1. private: data members and methods are accessible only inside the class.
2. public: data members and methods are accessible from any package.
3. protected: data members and methods are accessible within the same package and subclass (even in different package).
4. default: data members and methods are accessible within the same package only.

Using Access Modifiers with class:

* Outer class can only be made public and default(not specifying any modifier).
* Inner class can be any out of the 4 available access modifiers.
* Public class is accessible from anywhere in the program (in the same package as well as different packages)
* Default class is accessible only within the same package.
* A package can have multiple public classes, but a .java file can have only 1 public class



**CONSTRUCTORS**

* A constructor is a special member function used to initialize the objects of its class, at the time of object creation only.
* It is special because its name is same as the class name, and does not have a return type.
* Constructor is called only once(automatically), whenever an object of its associated class is created.
* If we do not define any constructor, then the compiler automatically supplies default implicit constructor.

Characteristics of Constructors:

* Can be declared in the private section.
* Do not have return types, not even void.
* Cannot be inherited, though a derived class can call the base class constructor using super() keyword.
* Can have default arguments.\*\*
* Cannot refer to their addresses.
* A constructor in Java can not be abstract, final, static, or synchronized(ssaf).

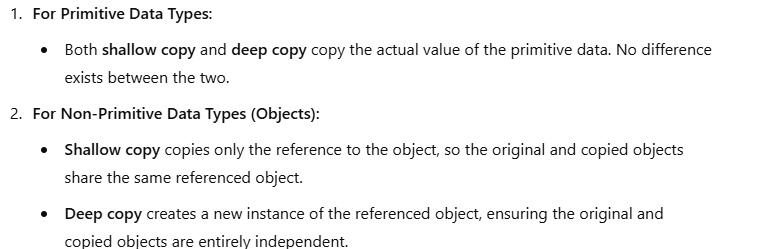
**synchronized:** The synchronized keyword in Java is used to prevent multiple threads from accessing critical sections of code simultaneously, ensuring thread safety. synchronization Is for shared resources and constructor is not shared like methods or variables, so it makes no sense to make a constructor synchronized.

**abstract:** if method is abstract then class will also be abstract, and an abstract class can’t be instantiated, meaning no object would be created. Also abstract methods need to be overriden by subclasses, but constructors are never inherited.

**TYPES OF CONSTRUCTORS**

1. Default Constructor: Constructor that does not take any arguments.
2. Parameterized Constructor: Constructor that can take arguments.
3. Copy Constructor(not provided by Java): Initializes an object using another object of the same class, at the time of object creation only. [Copy constructor](https://www.geeksforgeeks.org/copy-constructor-in-cpp/) takes a reference to an object of the same class as parameter.

We can create **shallow copy** as well as **deep copy** using copy constructor.



SHALLOW COPY USING COPY CONSTRUCTOR



DEEP COPY USING COPY CONSTRUCTOR

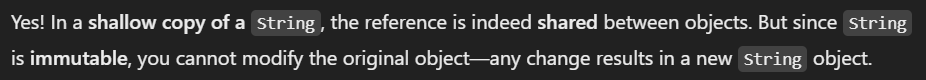


NOTE: The constructor's role is to initialize the object's state,after memory has been allocated. **Constructor doesn’t allocate memory for the object itself**. Memory allocation is managed by the compiler and the runtime environment.

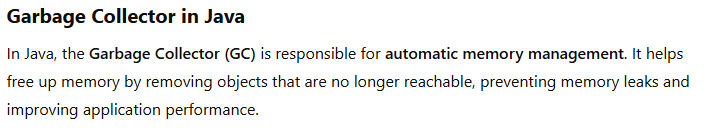
**CONSTRUCTOR OVERLOADING:** is when more than one constructor function is defined in a class.

***this* POINTER:** *this* is a special pointer that points to the current object.

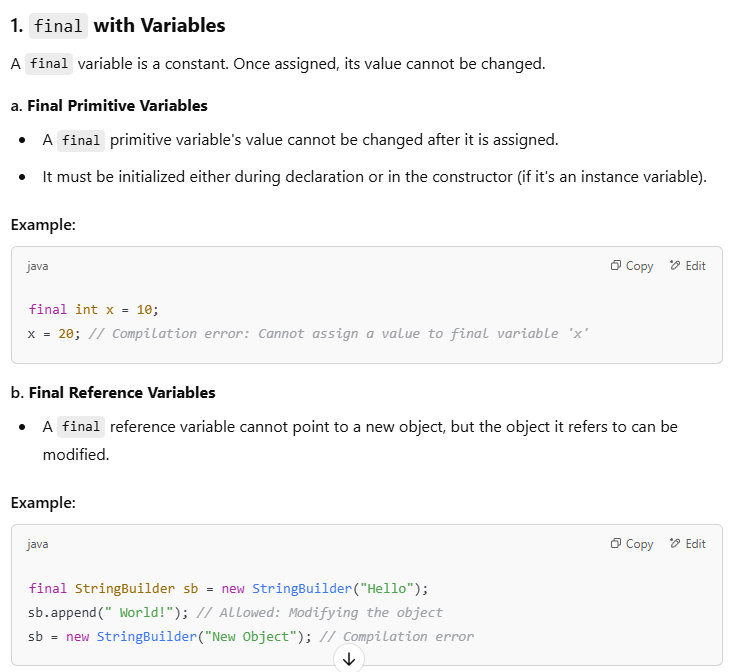
**SHALLOW COPY AND DEEP COPY:** There is no concept of shallow copy or deep copy in Strings or primitive data types.

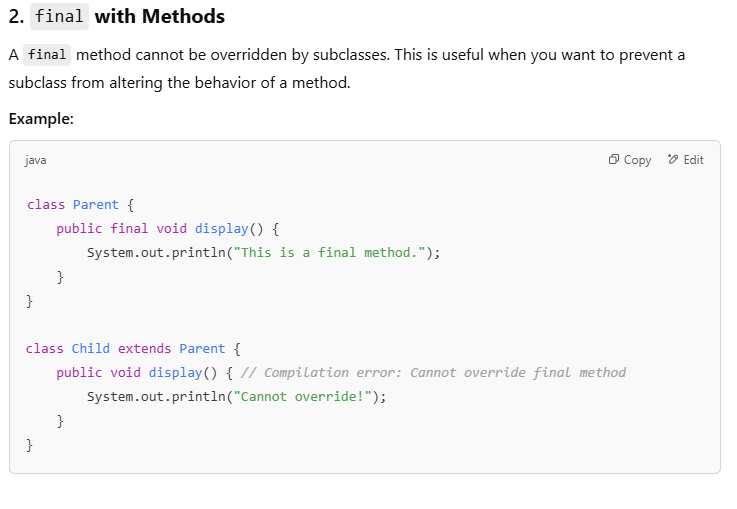


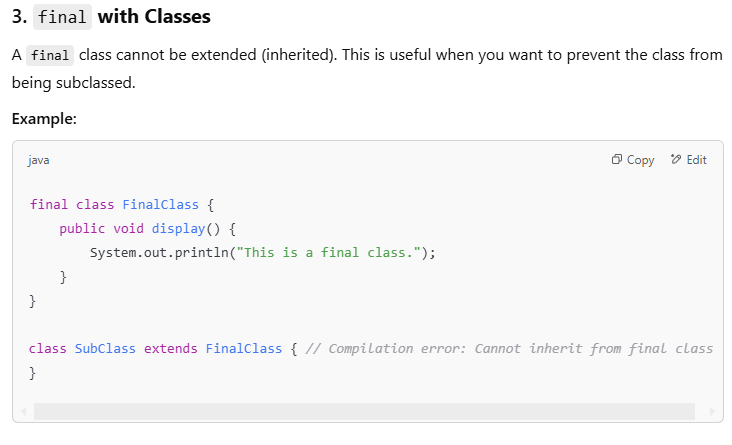
**GARBAGE COLLECTOR:** Deletes the unreferenced objects from the heap memory. System.gc() suggests the JVM to run garbage collector, but it is not guaranteed that it will run for sure.

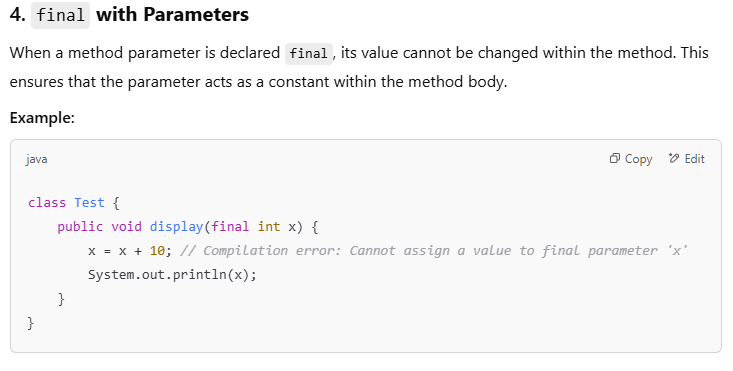


**final Keyword**

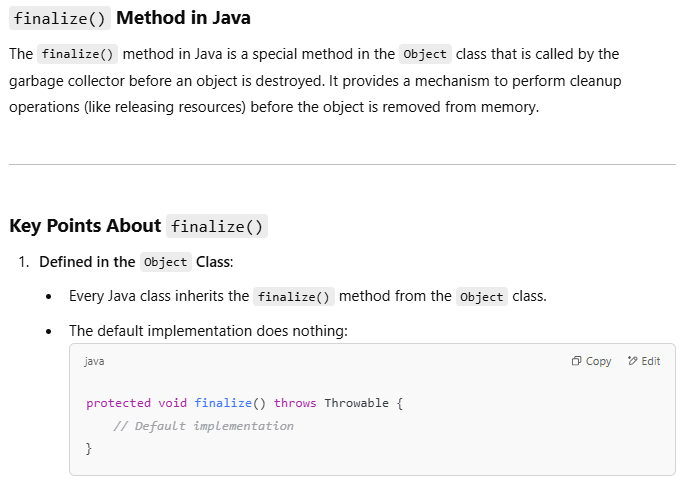


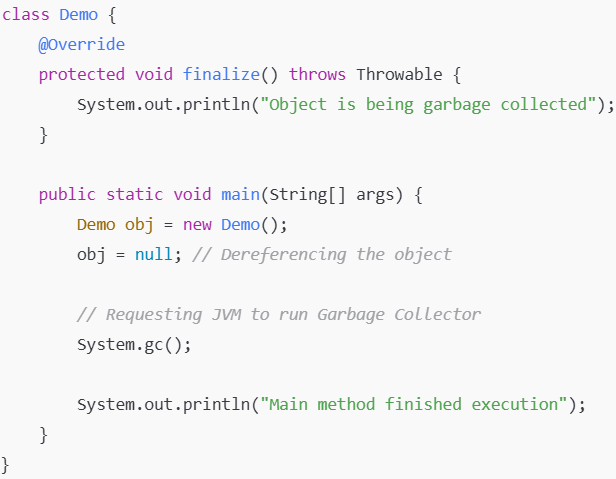


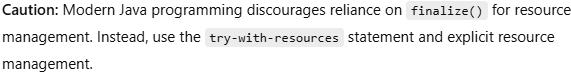




**finalize():**







**4 PILLARS OF OOPS**

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism
5. **ENCAPSULATION**

* The wrapping up of data members and methods inside a single unit,(class) and restricting direct access to data members, to ensure controlled interaction is called encapsulation. Encapsulation achieves data hiding.
* Member functions can be made public to access and manipulate these private data members.
* Encapsulation can be well understood by a real-life example of a capsule, it binds together different types of medicines in one single unit, i.e. capsule.

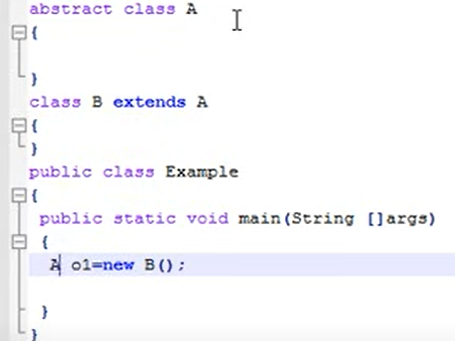
1. **ABSTRACTION**

* Abstraction refers to the act of representing essential features without including the background details or explanations.
* Abstraction in Java, can be achieved using abstract classes and interfaces(100%).
* Abstraction can be well understood by a real-life example. For Example, A man driving a car only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc. in the car.

**Abstract Class**:

* An abstract class is one that is not used to create objects, it only acts as a base class, to be inherited by other classes. A class can be made abstract using the abstract keyword.
* Abstract classes are used to declare common characteristics of sub classes. Eg- Person class can be declared abstract , and Student and Faculty class can inherit Person class.

NOTE: It is not possible to create objects of the abstract class, but reference of the abstract class can be created pointing to an object of a concrete subclass (a class that extends the abstract class and provides implementations for all abstract methods).



**Abstract Methods**(Data members can’t be abstract)

Abstract method is declared, without implementation in the abstract class, and must be overriden in the derived class. **It cannot be static**(static methods can’t be overriden), **final**(needs to be overriden) or **private**(must be accessible to child class for overriding).



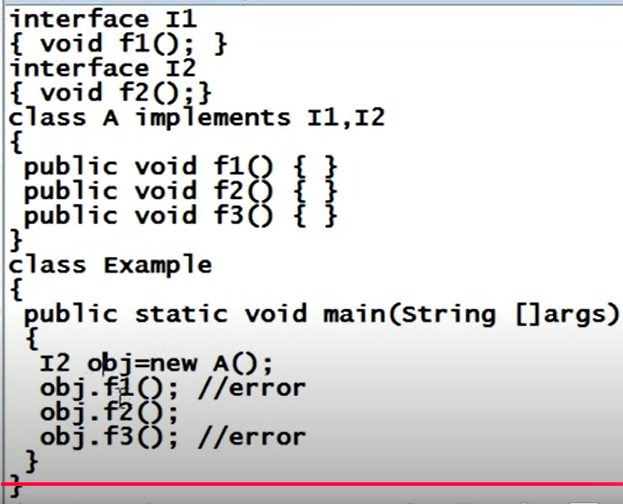
**INTERFACE:**

An interface in Java is a blueprint for a class that defines a set of methods that the implementing class must implement. It represents a contract that the implementing class agrees to adhere to, allowing Java to achieve multiple inheritance and 100% abstraction(prior to java 8, now static and default methods are also introduced).





Since later versions, methods can also be static, default and private;



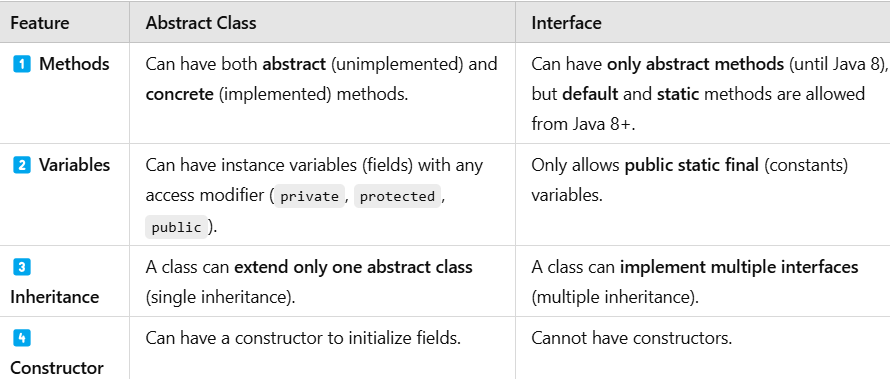
**REFERNECE OF THE INTERFACE**

**\* Reference of the interface can be created , and it can contain address of the object of any class that implements it. But it can call only the methods which are originally in that interface.**

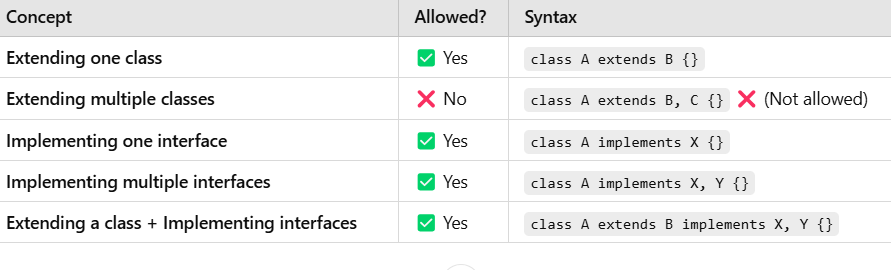
**SIMILARITIES B/W ABSTRACT CLASS AND INTERFACE:**

* Both abstract classes and interfaces cannot be instantiated directly. They serve as blueprints for other classes to implement or extend.
* Both are used to achieve abstraction.
* Both support polymorphism.
* Both can have abstract methods.

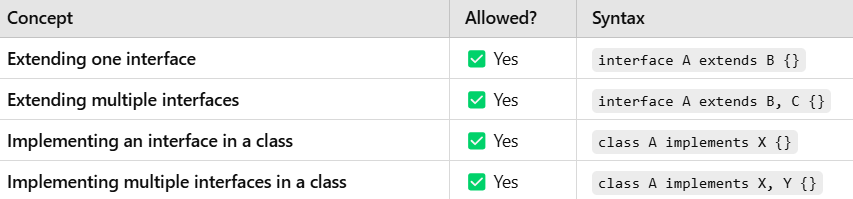
**DIFFERENCES B/W ABSTRACT CLASS AND INTERFACE:**

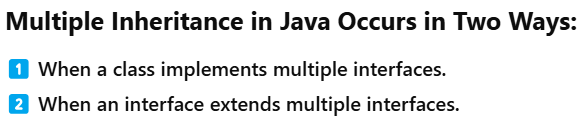


**Key Takeaways about Classes:**



**Key Takeaways about Interfaces:**

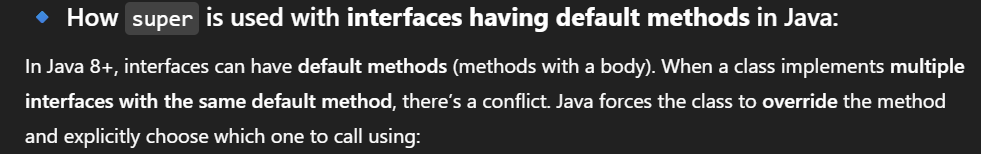


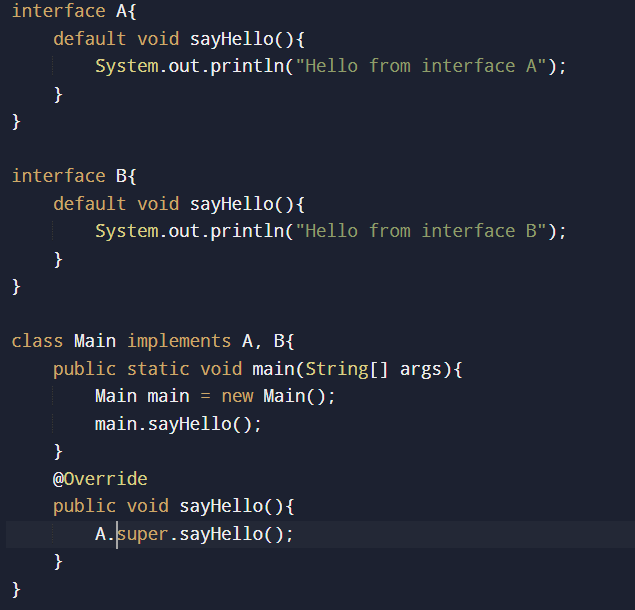


**Default Methods in Interface(Since Java 8)**

Default methods allow interfaces to provide default implementation for methods, for classes that implements them.

**Ambiguity Resolution when a class implements 2 interfaces, and each interface provides default implementation for a method - super**





**Static Methods in Interface(Since Java 8)**

* Just like default methods, they also contain complete definition. They can’t be overriden in the implementation class.
* Static methods inside interface can be called by using *interfacename.method()*, inside the implementation class.
* A case may arise when you may observe that implementation class is overriding a static method of the interface, actually it is not overriding. The static method of the interface is *not visible* to the implementation class, and it is declaring its own separate method.
* The psvm method, which we usually write inside Main class can also be written inside an interface, due to Java 8 feature.

1. **INHERITANCE**

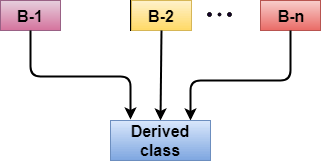
* Inheritance allows the objects of one class to acquire the properties and behaviors of another class. The class which acquires the properties and behaviors is called the derived class, and the class from which it acquires, is called the base class.
* Inheritance provides re-usability of code, additional features can be added to existing class without modifying it.

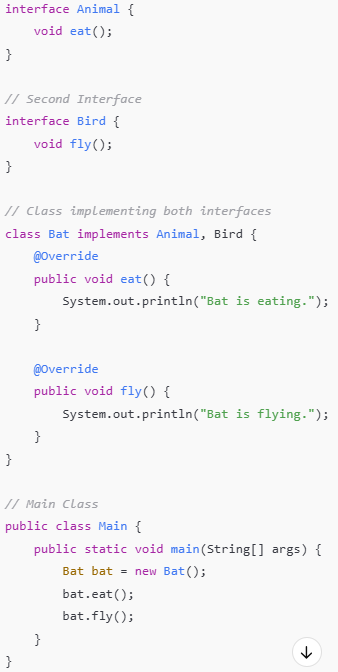
TYPES OF INHERITANCE

1. Single Inheritance
2. Multiple Inheritance (only through interfaces)
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance (only through interfaces)
6. SINGLE INHERITANCE: A single derived class inherits from only one base class.

C++ Inheritance

1. MULTIPLE INHERITANCE: When one class implements multiple interfaces, or one interface extends multiple interfaces

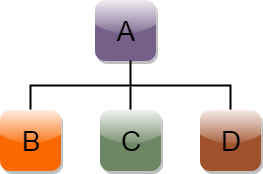




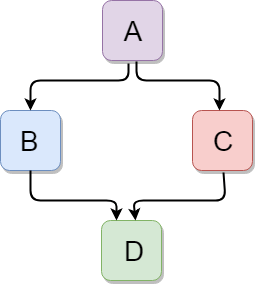
1. MULTILEVEL INHERITANCE: When one derived class inherits from another derived class.

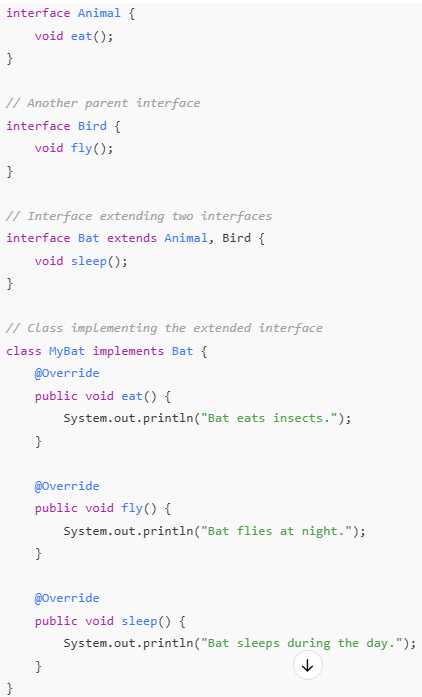
C++ Inheritance

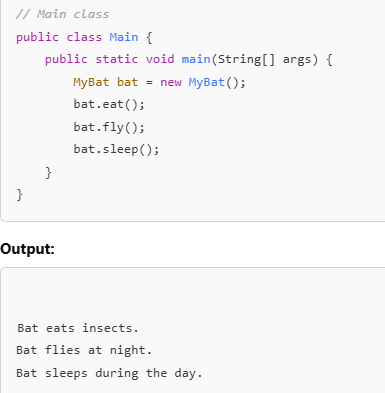
1. HEIRARCHICAL INHERITANCE: When multiple derived classes inherit from single base class.



1. HYBRID INHERITANCE: Combination of more than one type of inheritance.







NOTE:

* Parent class object can’t call child class data members/methods, but reference variable of the parent class can be created to point to objects of the child class.
* When parent class reference variable is used to create object of the child class, then type of the reference variable (parent class) will be taken into consideration, for calling data members/ methods. (no overriden method present in class)
* But if a method is overriden in the child class, then actual type of the object to which reference variable is pointing, will be used to determine which version of the function to call. (dynamic method dispatch)

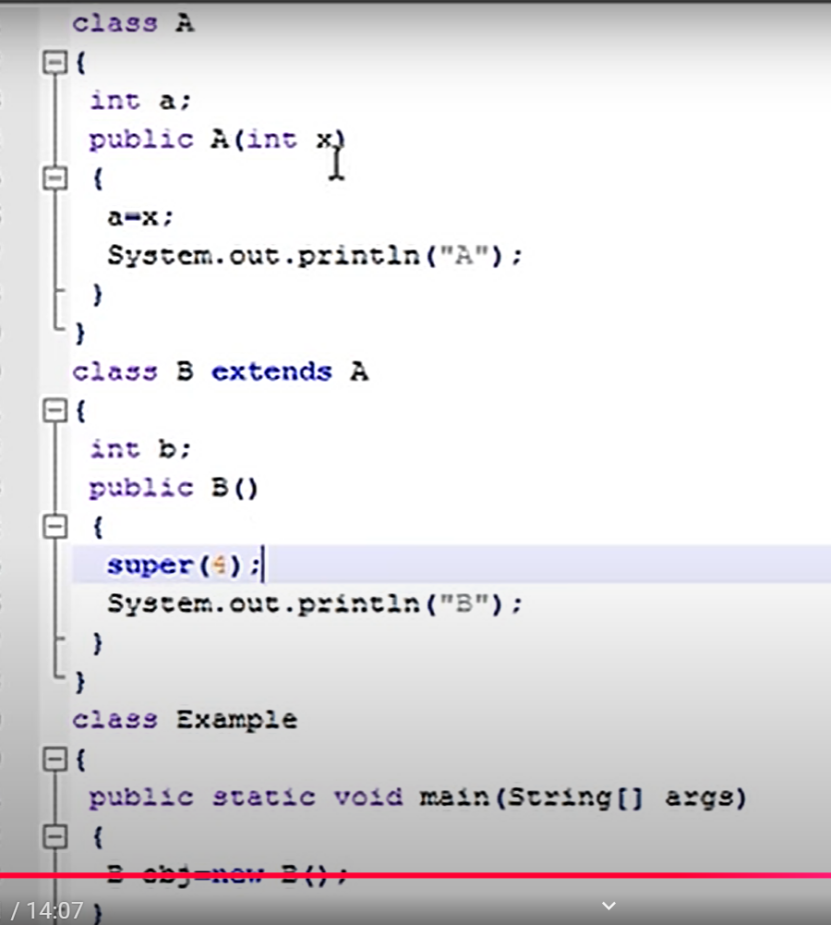
**CONSTRUCTOR IN INHERITANCE:**

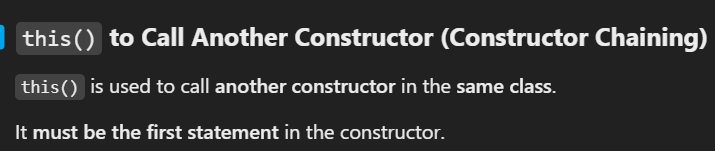
The first line in any constructor is super(), no matter we write it or not. It calls the default constructor of the base class. Even if the class does not extends any class, by default it extends the Object class.

Order of Execution: Base class Constructor, then Derived class Constructor

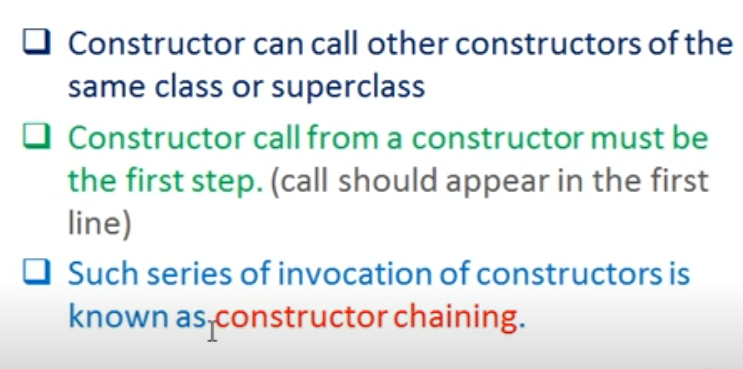
Order of Call: Derived class Constructor, then Base class Constructor(called by derived class)

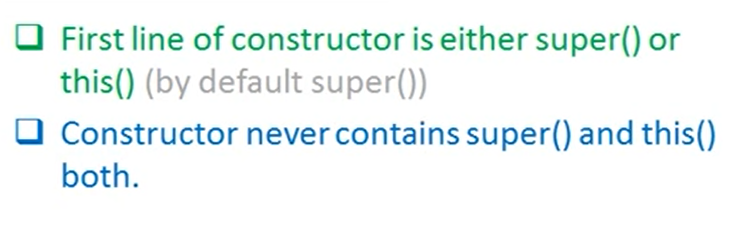
Base class **default** constructor is called automatically by the derived class constructor, but if the base class does not have any default constructor, then we explicitly need to call the base class constructor in the derived class constructor, by using super().(mandatory)



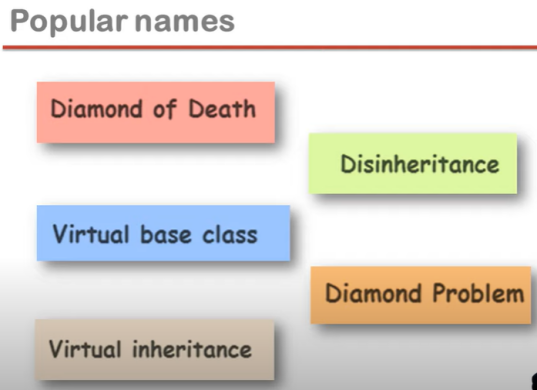


**CONSTRUCTOR CHAINING**

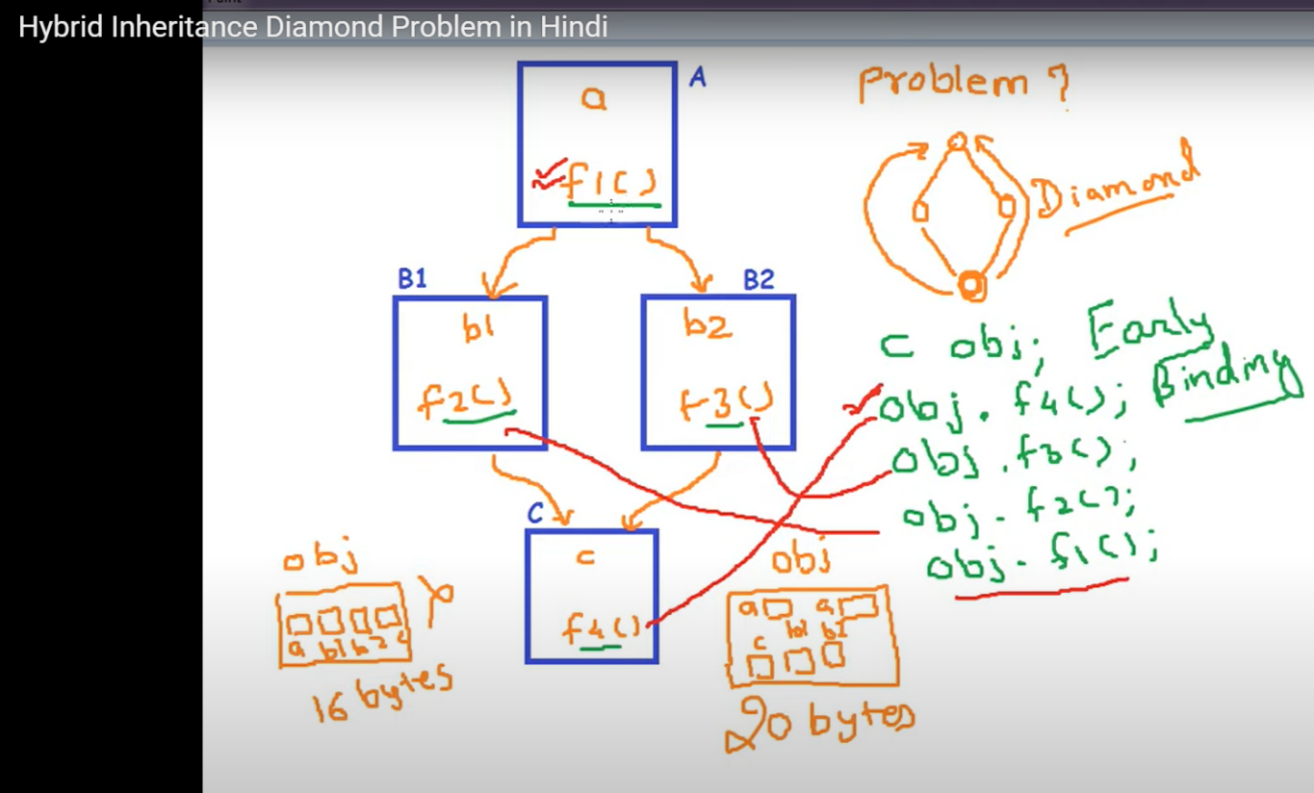




**VIRTUAL BASE CLASS**:

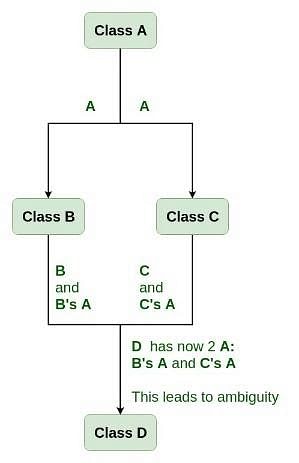


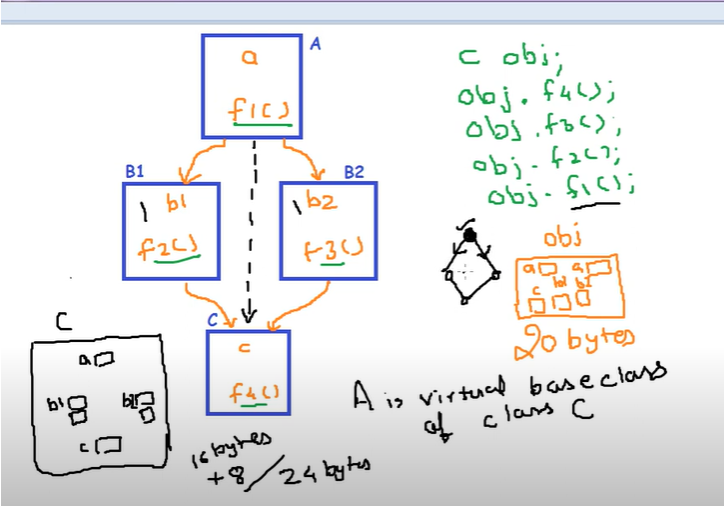
Hybrid Inheritance Diamond Problem:



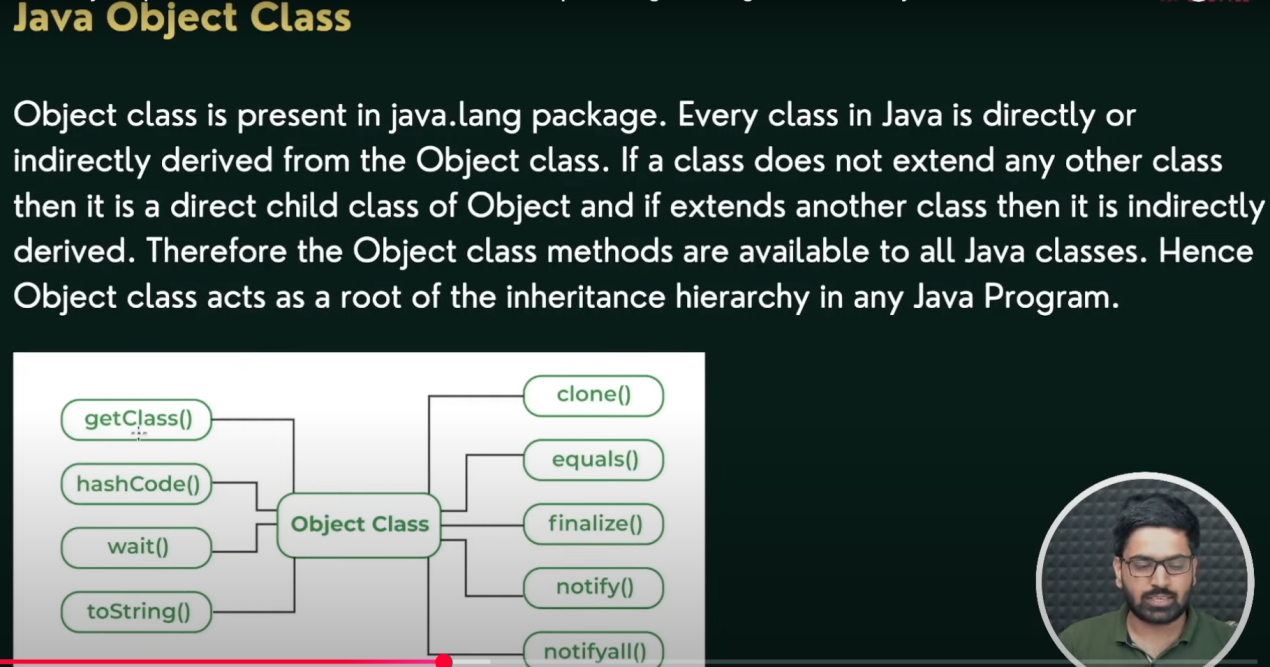


**Due to hybrid inheritance, a situation may arise, where the derived class can have properties of one base class more than once. To solve this, we use virtual base classes to ensure only one shared instance of the base class exists in the inheritance chain.**

****



**JAVA OBJECT CLASS**

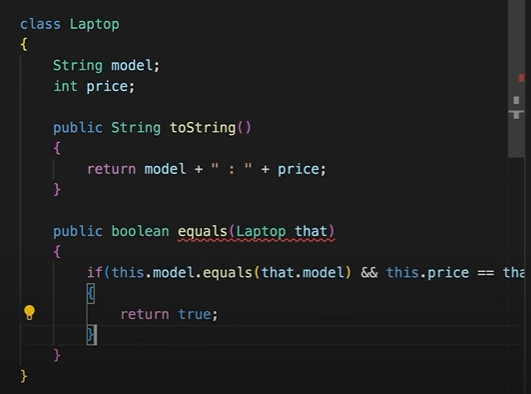


toString(): provides a string representation of an object. Whenever we try to print reference of an object using sout, toString() method is called. Default implementation returns *ClassName@hashcode.* We can override it.

equals(): if we use == to compare 2 string objects, it will compare the reference, not the content. equals() compare the content of two strings. For other objects equals() also compares the reference, we can override it for content comparison.

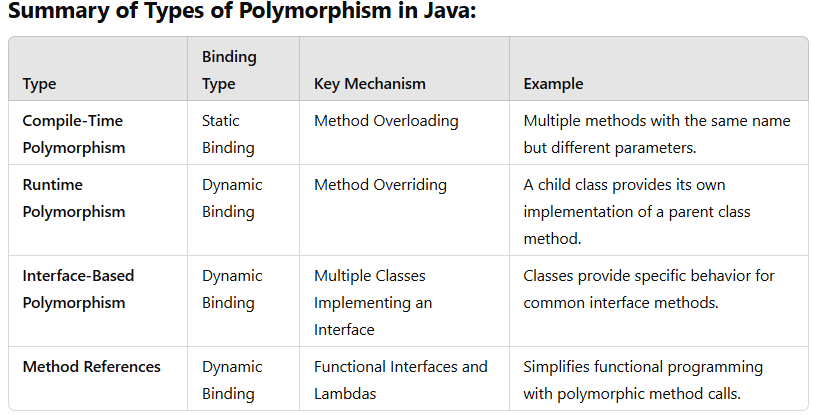
getClass(): final method that is used to obtain run time class of an object.

*public final Class<?> getClass();*



1. **POLYMORPHISM**

* Polymorphism simply means one name having multiple forms. The word ‘Polymorphism’ has been derived from the Greek words ‘poly’ which means many and ‘morphe’ which means forms.
* It provides code reusability. Polymorphism can be better understood with the help of a real life example, a person can be a father, son, employee or friend at the same time. The same person acts differently in different situations.



**COMPILE TIME POLYMORPHISM:**

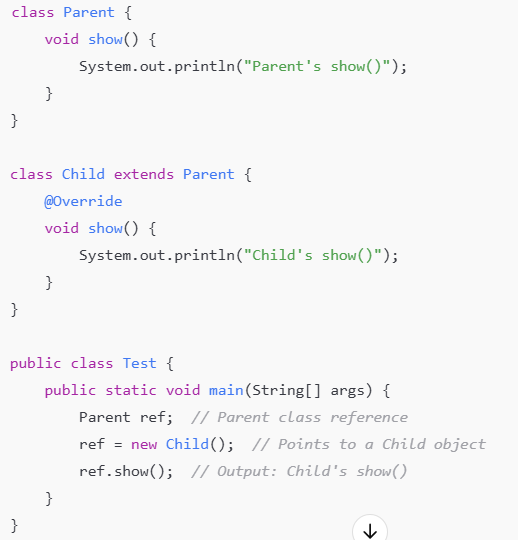
* Also known as Static polymorphism or Early Binding.
* Can be achieved through method overloading.
* Decision about which method to call is made at compile time, based on the reference type(info available at compile time).

**METHOD OVERLOADING:**

* When there are multiple methods in same class with the same name but different parameters, these methods are overloaded.
* Methods can be overloaded by using:
* Diff no of parameters
* Diff types of parameters
* Based on the sequence of parameters

**RUNTIME POLYMORPHISM:**

* Also known as dynamic polymorphism , late binding or dynamic method dispatch and is achieved only when there is **method overriding.**
* It can be defined as the ability of a parent class reference variable to point to objects of its child class, and at runtime determine which version of a method to call based on the actual type of the object to which the reference is pointing, and not on the type of the reference.
* Runtime polymorphism is achieved when the method in the parent class is overridden in the child class.
* If there is no method overriding, then type of the reference variable will be considered, which is not runtime polymorphism.



**Method Overriding and:**

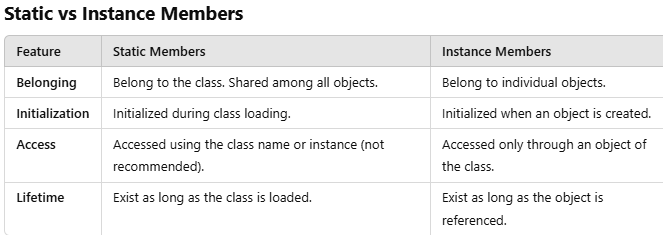
**Method overriding:**

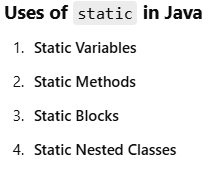
* Method overriding occurs when a child class provides its own implementation of a parent class method, while keeping the method signature (name, parameters, and return type) the same.
* The return type of the overriding method can be covariant return type.
* Method overloading is resolved at runtime.
* Static methods cannot be overriden, and do not exhibit runtime polymorphism.

**Method hiding:**

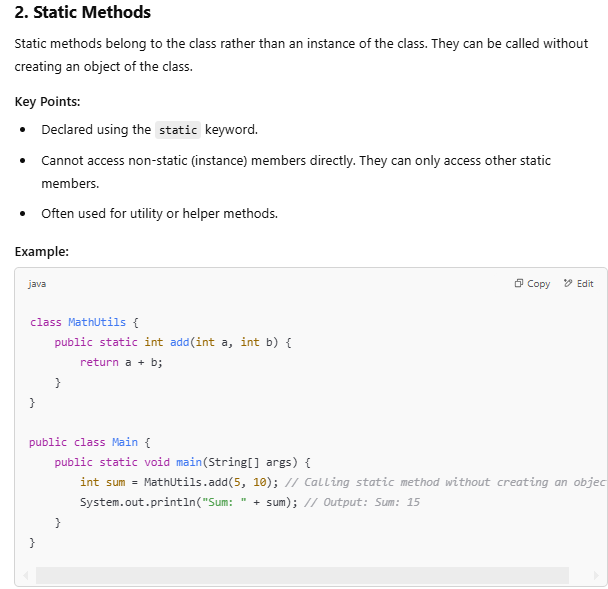
* Method hiding is an extension of method overriding only where the method which is being overriden and the method which overrides both are static.
* The return type of both static methods must be exactly same.
* Unlike method overloading, method hiding is resolved at compile time, which means that if parent class reference points to child class object, then reference type will be taken into consideration while making a call to overriden method.

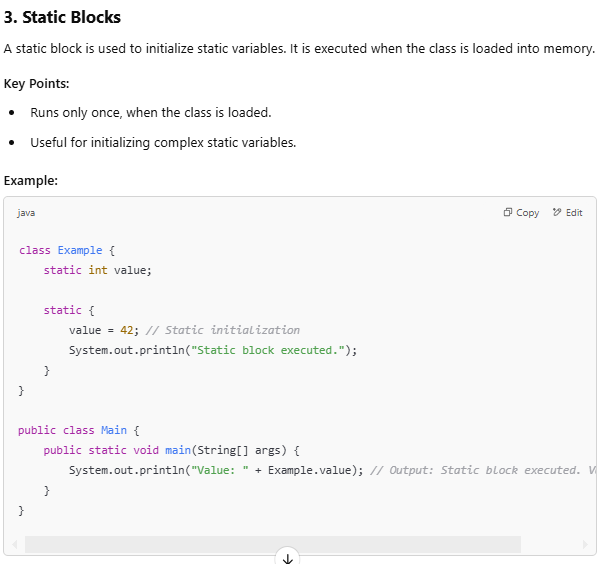
(BILA)





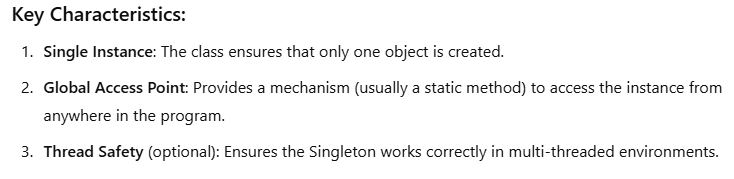




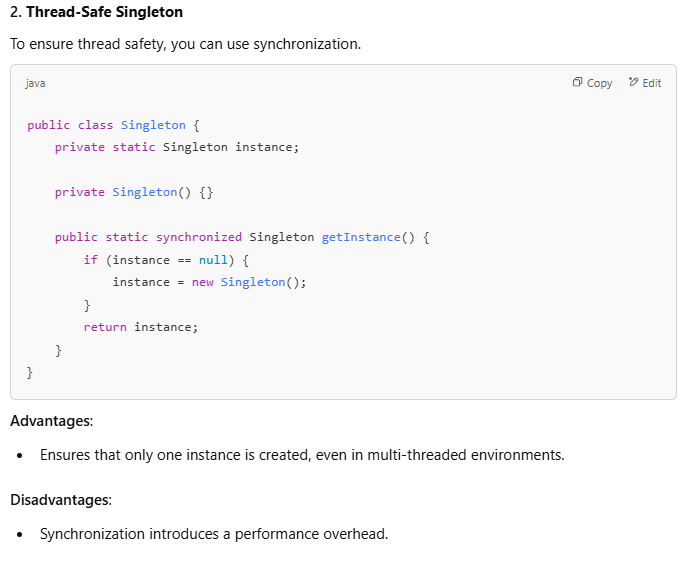


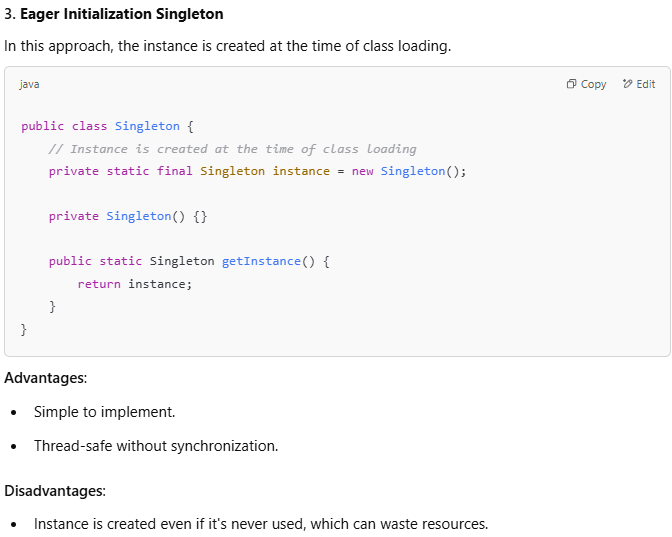


**SINGLETON CLASS**











**JAVA EXCEPTION HANDLING**

**Exception:** An exception is an unexpected event that occurs during program execution and disrupts the normal flow of instructions.

**Types Of Exceptions:**

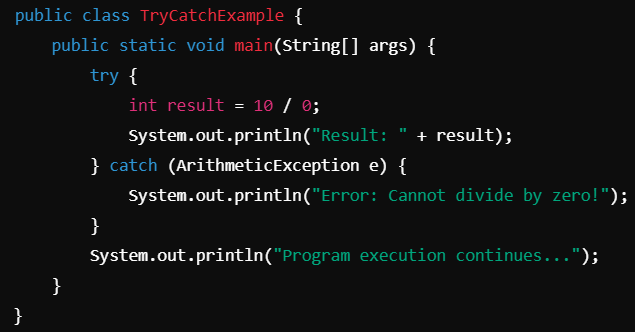
1. **Java Runtime Exceptions/Unchecked Exceptions**

* Occurs due to programming error (fault of the programmer).
* These exceptions are not checked at compile time but at run time.
* Eg- accessing array index out of bond- *ArrayIndexOutOfBoundsException*, dividing a no by 0- *ArithmeticException*, or null pointer access- *NullPointerException*. All these are classes which extend the Exception class.

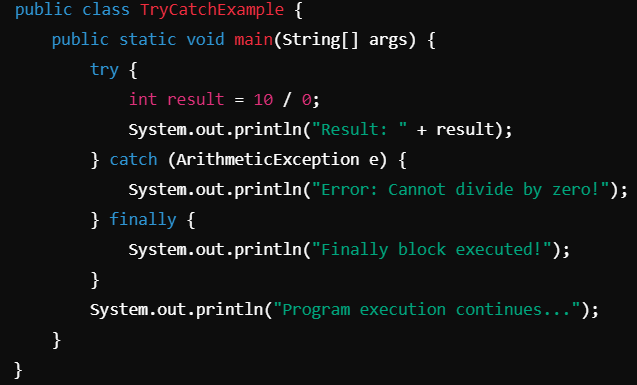
1. **Java IO exception/Checked Exception**

* These exceptions are checked at compile time, and the programmer is prompted to handle these exceptions.
* Eg- Trying to open a file that does’nt exists- *FileNotFoundException*, Trying to read past the end of a file.

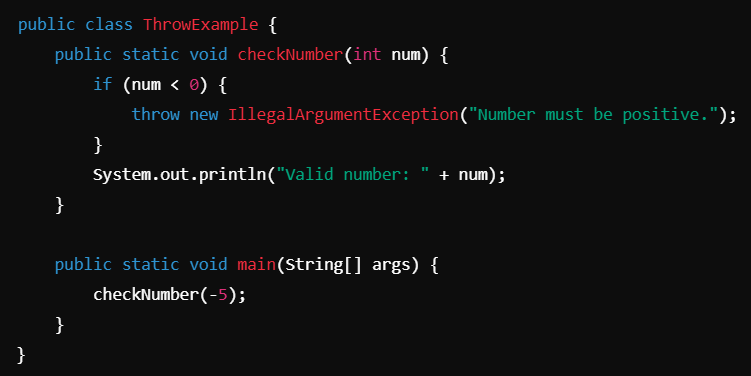
**try-catch block:** We can add multiple catch blocks, to handle more than one exception for a single try block.



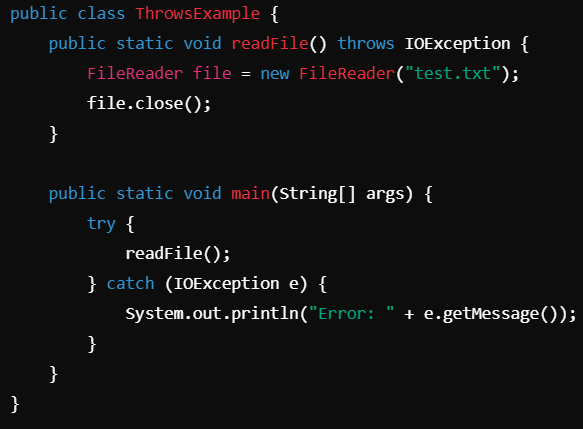
**try-catch-finally:** finally block always gets executed, no multiple finally blocks allowed.



**throw:** used to manually throw an exception in java



**throws:** is used to declare exceptions that a method might throw. It tells the caller that an exception may occur, but it does not handle the exception itself.



**How Java Code Executes:**

1. Java programs are written, and saved with the .java extension.
2. The javac compiler compiles the .java file into a .class file containing bytecode, which is an intermediate, platform-independent code that can be executed by the Java Virtual Machine (JVM).
3. Once compiled, the class loader in JVM loads the .class file into memory.
4. The Bytecode Verifier in JVM performs checks to ensure:

* No unauthorized memory access occurs.
* No illegal type conversions are made.
* No attempts are made to bypass Java’s security mechanisms.

1. JVM Executes the Bytecode Using Interpreter or JIT Compiler

Interpreter converts bytecode to machine code line by line. JIT compiler converts entire sections of bytecode into native machine code at once. JVM switches to JIT compilation after detecting frequently executed code (hotspots).

**JDBC**

**SQL:** Language used to manage and query databases.

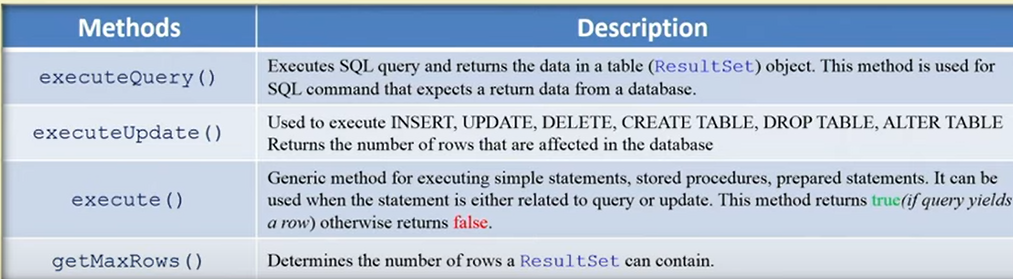
**MySQL:** A DBMS that uses SQL.

**java.sql package:** contains classes and interfaces of JDBC.

3 classes - **java.sql.DriverManager**, **ResultSet and SQLException**

3 interfaces - **java.sql.Driver, java.sql.Connection, java.sql.Statement**

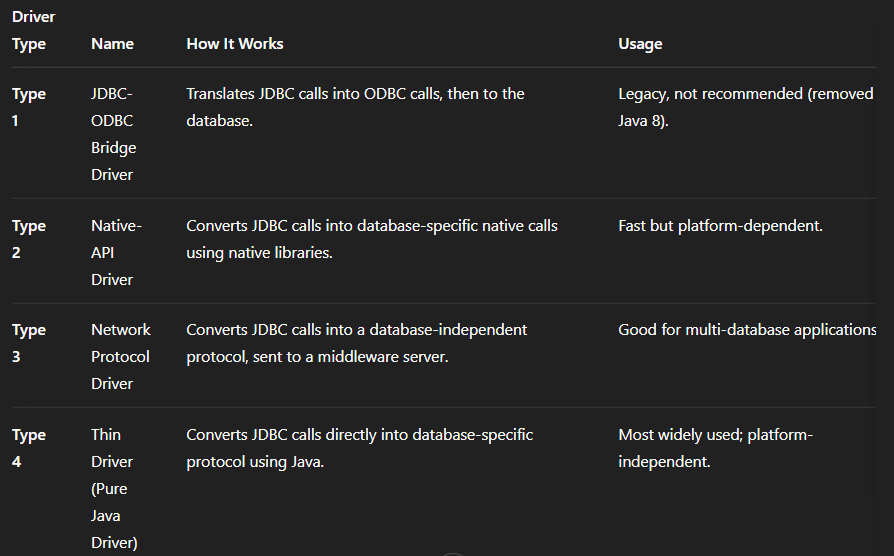
**JDBC Driver:** Java API that Connects Java Application to Database



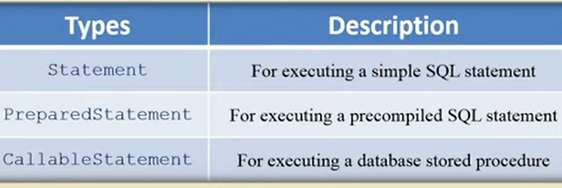
execute(): for queries that might return more than 1 resultSet.

executeUpdate(): can be used with DDL as well as DML

**Types Of JDBC Drivers:**



**3 Types of Statement:**



**Change MySQL Password:**







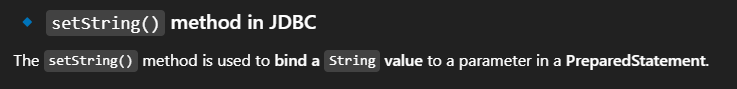
**JDBC INITIAL CODE:**

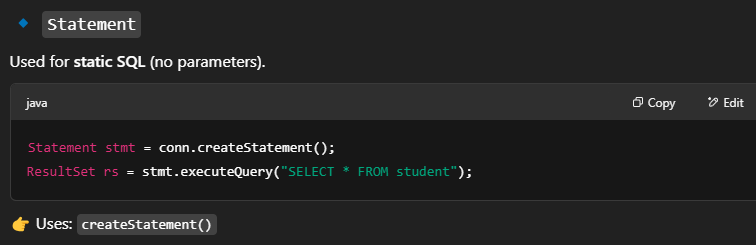
**executeQuery() method** of java.sql.Statement interface is used to execute SELECT statement.

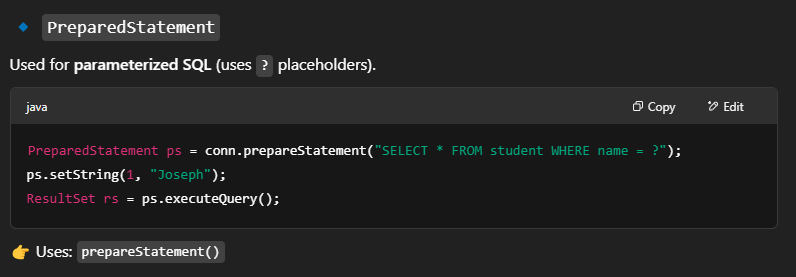
**executeBatch() method** executes batch of statements and return the number of rows affected.

**DriverManager class** manages connections, it has a static method getConnection().

**getWarnings() method** retrieve warnings occurred during database access.







**Correct Order for closing the resources:**

Result Set -> Statement -> Connection

A screenshot of a computer program

AI-generated content may be incorrect.

**7 Steps:**

1. Import package: java.sql.\*
2. Load(from JAR file) and Register the driver: *com.mysql.jdbc.Driver, forName()-register*
3. Establish the connection: Instantiate Connection Interface
4. Create the statement
5. Execute the query
6. Process Results
7. Close the connection