

Java

SURAJ P BHADANIA

Java syntax:

- structure of java includes classes, methods, and statements.
- java is case sensitive

Java Output:

- to print in java we use `System.out.println();`
- here System is final class in java.lang package , out is an instance in `PrintStream` class , and `println` is method in `PrintStream`.

Java Comments:

- to comment for single line we can use `//`
- for multiple lines we can use `/* ... */`.

Java Variables:

- variables must be declared with specific type such as `int` , `double` , `String` and so on.
- variables can be initialized upon declaration.
- must begin with a letter (A to Z or a to z), currency character (`$`), or an underscore (`_`).

Java Datatypes:

- primitive : includes `int` , `byte` , `short` , `long` , `float` , `double` ,`char`
- non primitive : includes objects , arrays

Java typecasting:

- we can convert datatypes in java like convert `int` to `double` using typecasting.

Java Operators:

- arithmetic : +,-,*,/,% for mathematical operations
- logical : ==,!=,>,<,&&,|| for comparison

Java Strings:

- Strings are considered as objects in java
- has methods like length() , charAt(), substring() , concat() and many more.

Java Math:

- math class provides many methods to perform basic operations like max() , min() , sqrt()

Java Boolean:

- Boolean type has two values true and false
- used in performing logic comparisons like AND and OR.

Java if else:

- are conditional statements used to perform different actions based on different conditions

- Syntax for this is ,

```
if(condition){  
    //code to be executed if condition true.  
}  
else{  
    // code to be executed if condition false.  
}
```

- can have multiple statements by using elseif.

Java Switch:

-same as if else statements but in a cleaner way.

-Syntax:

```
switch( ){  
    case :  
        break;  
    case:  
        break;  
    default:  
}.
```

Java While loop:

-represents a block of code until the condition is true.

-while(condition){

```
    //code to be executed
```

```
}
```

Java For loop:

-same as while loop but we can repeat block of code for specific number of times.

-syntax

```
for(initialization; condition; update){
```

```
    //code to be executed
```

```
}.
```

Java Break/Continue:

-adding break in while or for loop exits the loop immediately skipping remaining code.

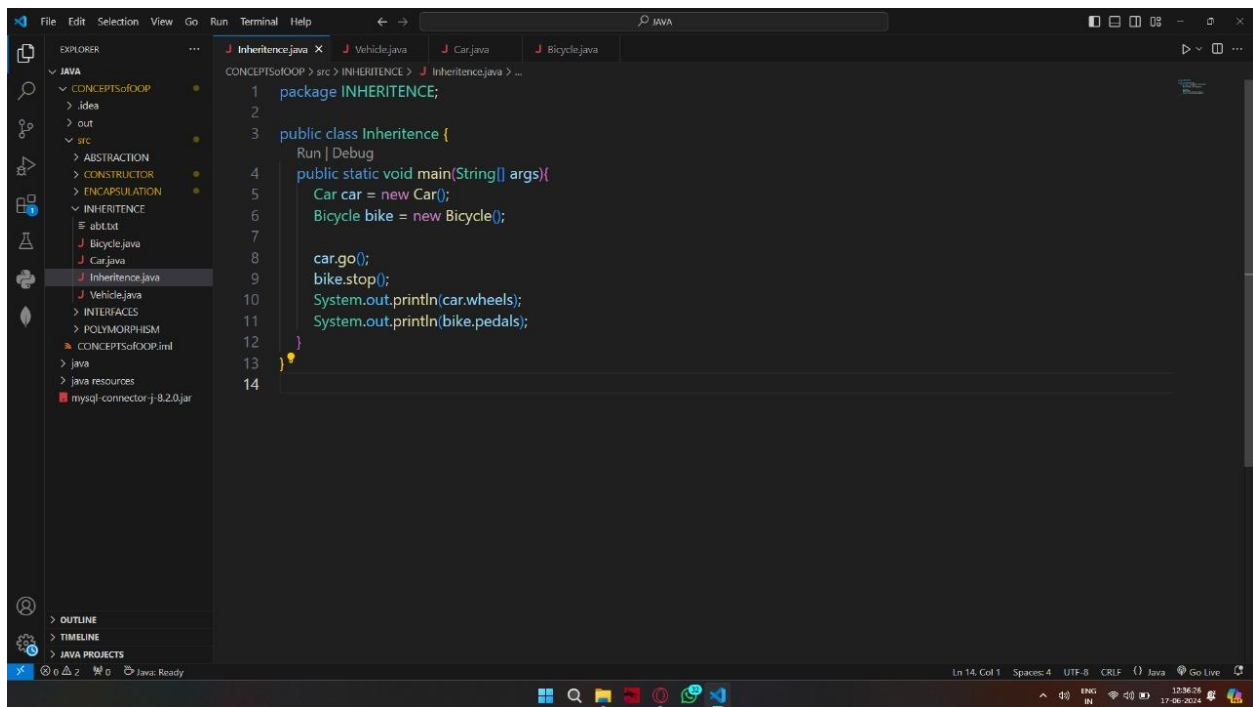
-continue statement skips the current iteration of the loop.

OOPS

Java Inheritance:

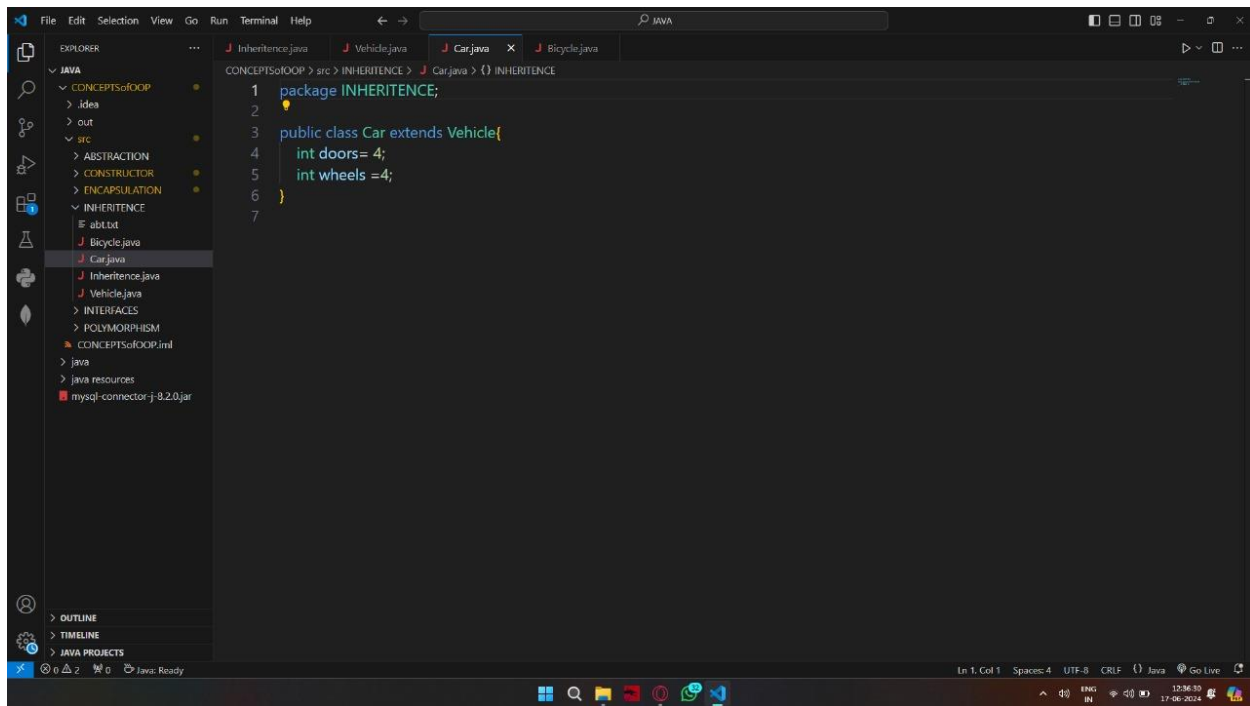
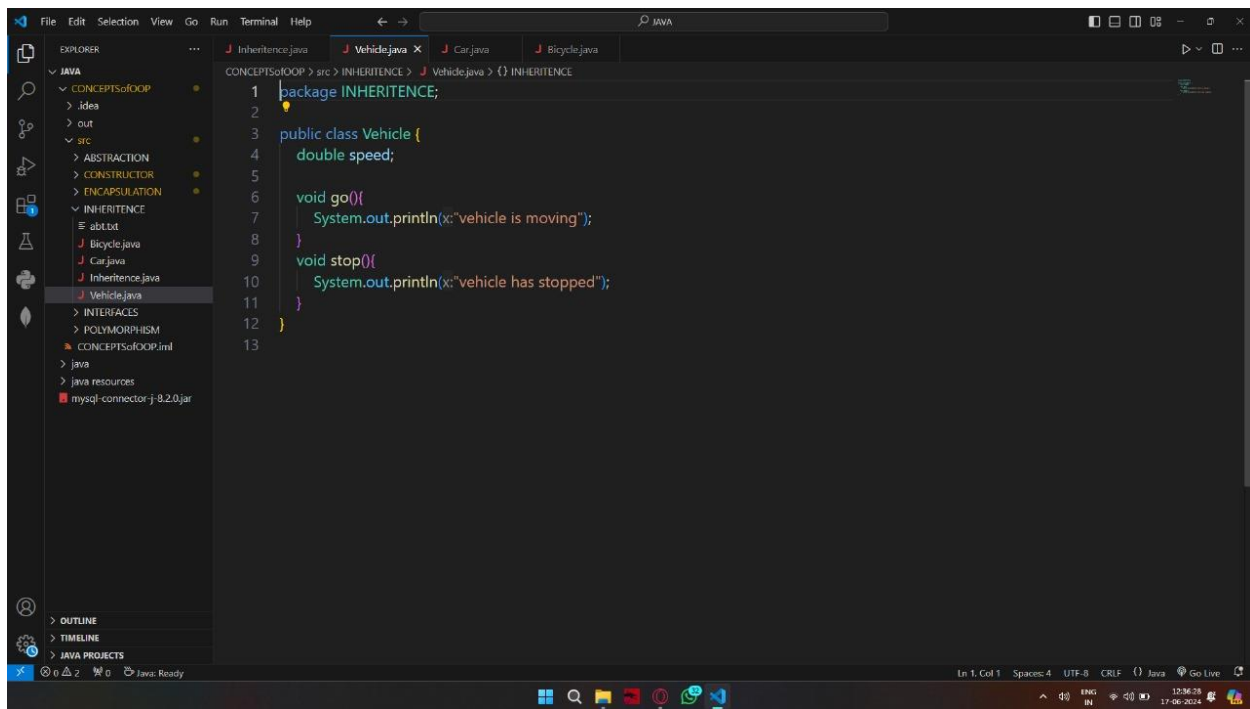
- mechanism to inherit methods and attributes of one class to another class.
- inherited class is known as child class and from which it inherited is parent class.
- multiple inheritance is not allowed in java.

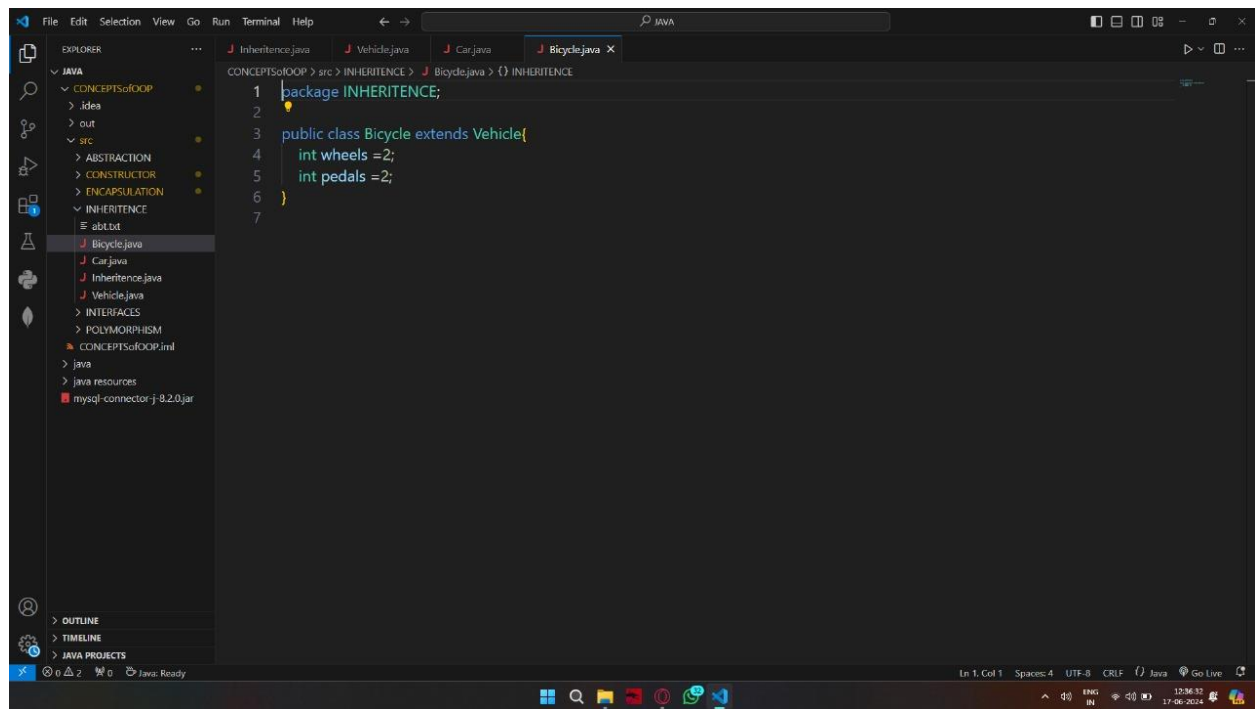
Ex:



The screenshot shows an IDE window with a project named 'CONCEPTSOOP'. The 'EXPLORER' sidebar on the left shows a directory structure with folders for 'CONCEPTSOOP', 'src', 'ABSTRACTION', 'CONSTRUCTOR', 'ENCAPSULATION', 'INHERITANCE', 'INTERFACES', and 'POLYMORPHISM'. Under the 'INHERITANCE' folder, there are files named 'Bicycle.java', 'Car.java', and 'Inheritance.java'. The 'Inheritance.java' file is selected and its code is displayed in the main editor. The code defines a package 'INHERITENCE' and a public class 'Inheritance' with a 'main' method. Inside the 'main' method, instances of 'Car' and 'Bicycle' are created and their 'go' and 'stop' methods are called, with their attributes ('wheels' and 'pedals') printed to the console.

```
1 package INHERITENCE;
2
3 public class Inheritance {
4     Run | Debug
5     public static void main(String[] args){
6         Car car = new Car();
7         Bicycle bike = new Bicycle();
8
9         car.go();
10        bike.stop();
11        System.out.println(car.wheels);
12        System.out.println(bike.pedals);
13    }
14 }
```





Java Polymorphism:

- poly means 'many' , morphism means 'forms' i.e many forms
- method in child class **OVERRIDES** methods in parent class , i.e we are now doing same thing in a different form
- ability of an obj to identify as more than one type.

Ex:

This screenshot shows an IDE window with the file `Polymorphism.java` open. The Explorer on the left shows a project structure with a `POLYMORPHISM` package containing `Animal.java`, `Cat.java`, `Eagle.java`, and `Tortoise.java`. The main editor displays the following code:

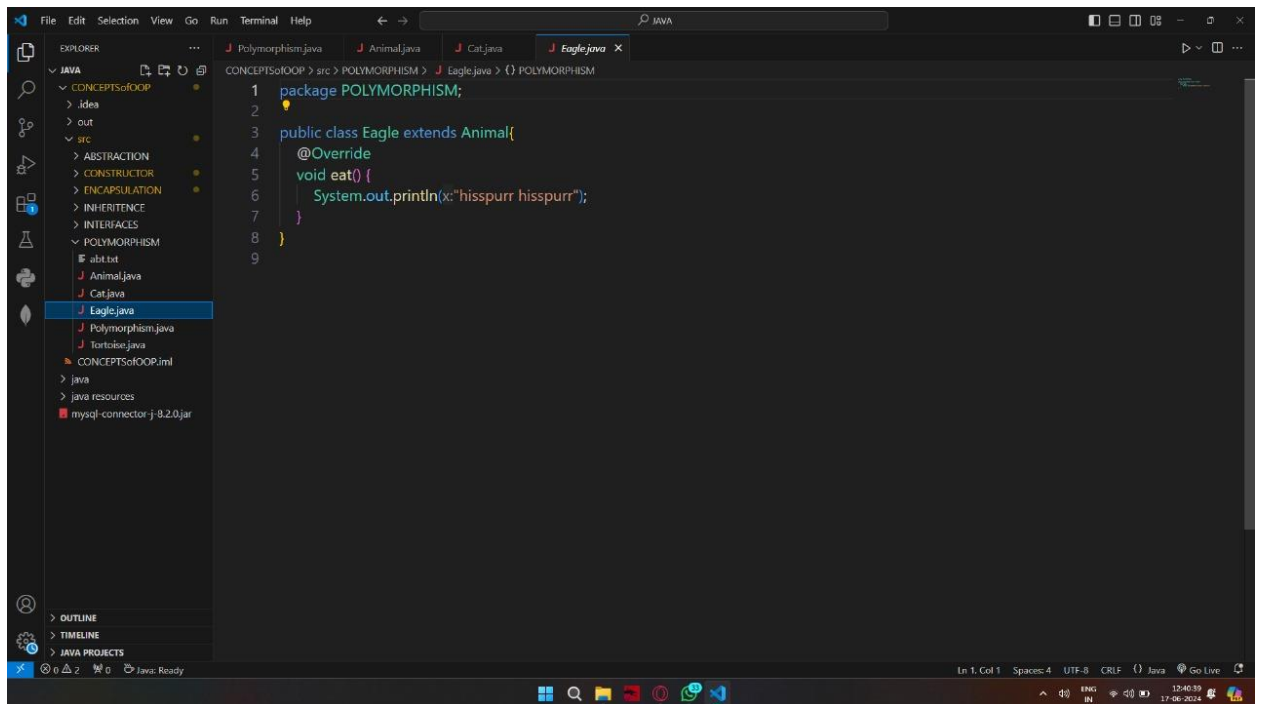
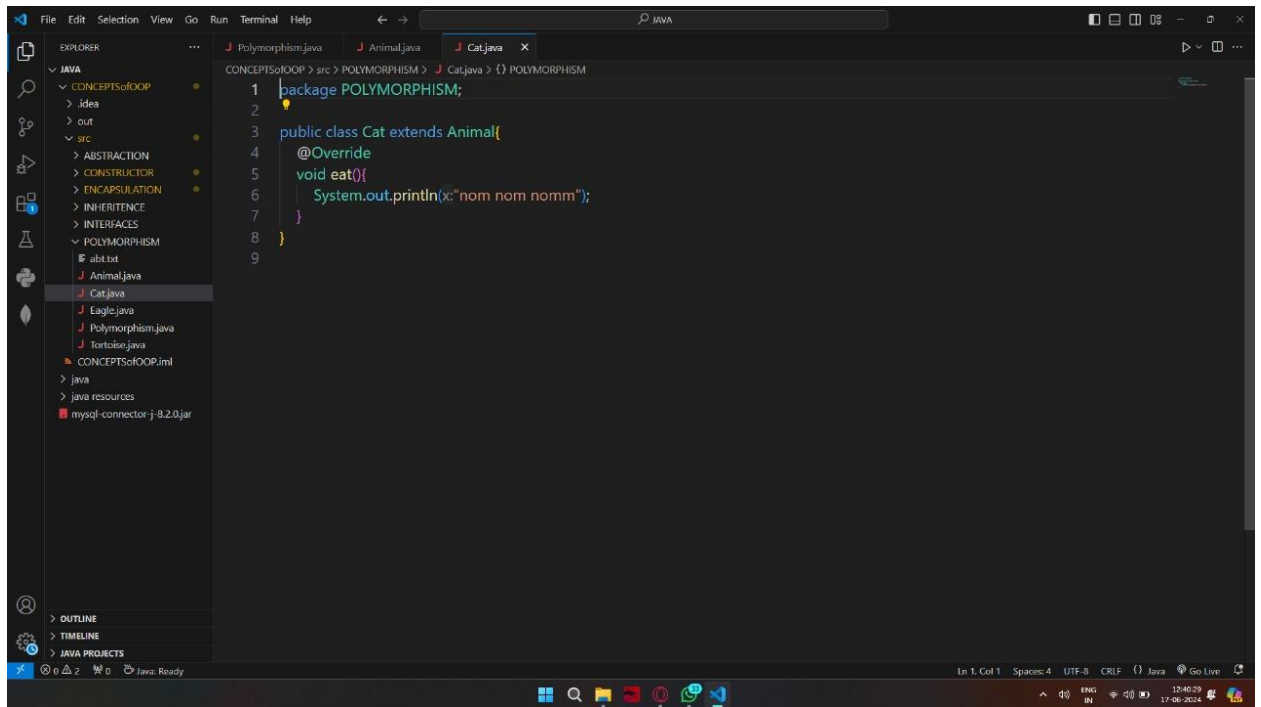
```
1 package POLYMORPHISM;
2
3 public class Polymorphism {
4     Run | Debug
5     public static void main(String[] args){
6         Eagle eagle =new Eagle();
7         Tortoise tortoise =new Tortoise();
8         Cat cat=new Cat();
9
10        Animal[] species =(eagle,tortoise,cat);
11
12        for(Animal x : species)
13            x.eat();
14    }
15 }
```

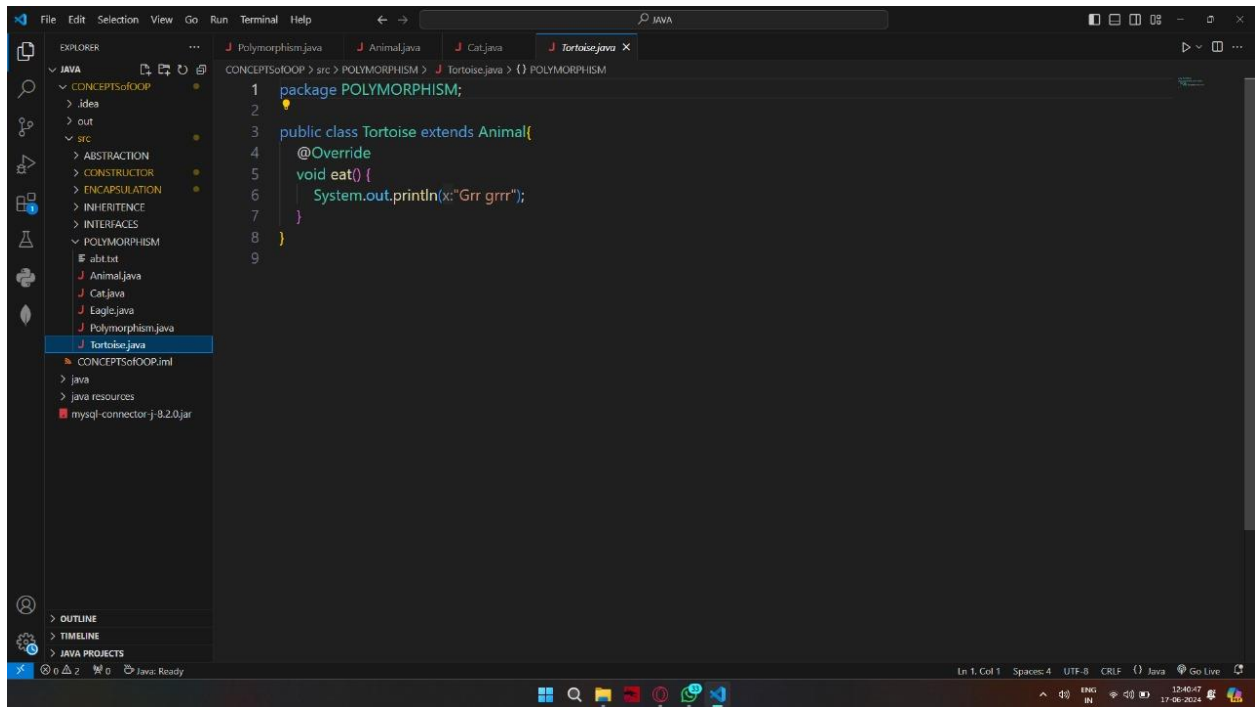
The status bar at the bottom indicates the cursor is at line 1, column 1, with 4 spaces, UTF-8 encoding, and CRLF line endings.

This screenshot shows the same IDE with the file `Animal.java` open. The Explorer on the left now highlights `Animal.java`. The main editor displays the following code:

```
1 package POLYMORPHISM;
2
3 public class Animal {
4     String name;
5     int age;
6
7     void eat(){
8         System.out.println(x:"munch munch");
9     }
10 }
11 }
```

The status bar at the bottom indicates the cursor is at line 1, column 1, with 4 spaces, UTF-8 encoding, and CRLF line endings.





In the above example , I have explained method overloading and method overriding.

Java Abstraction:

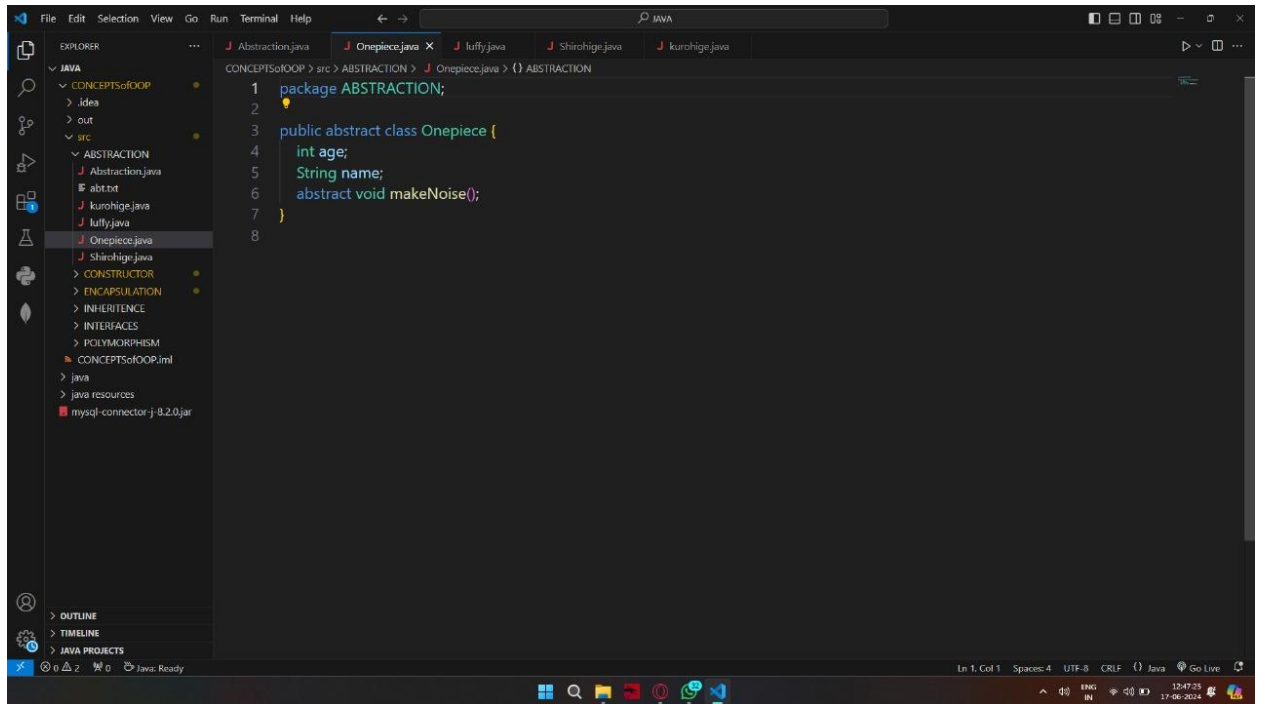
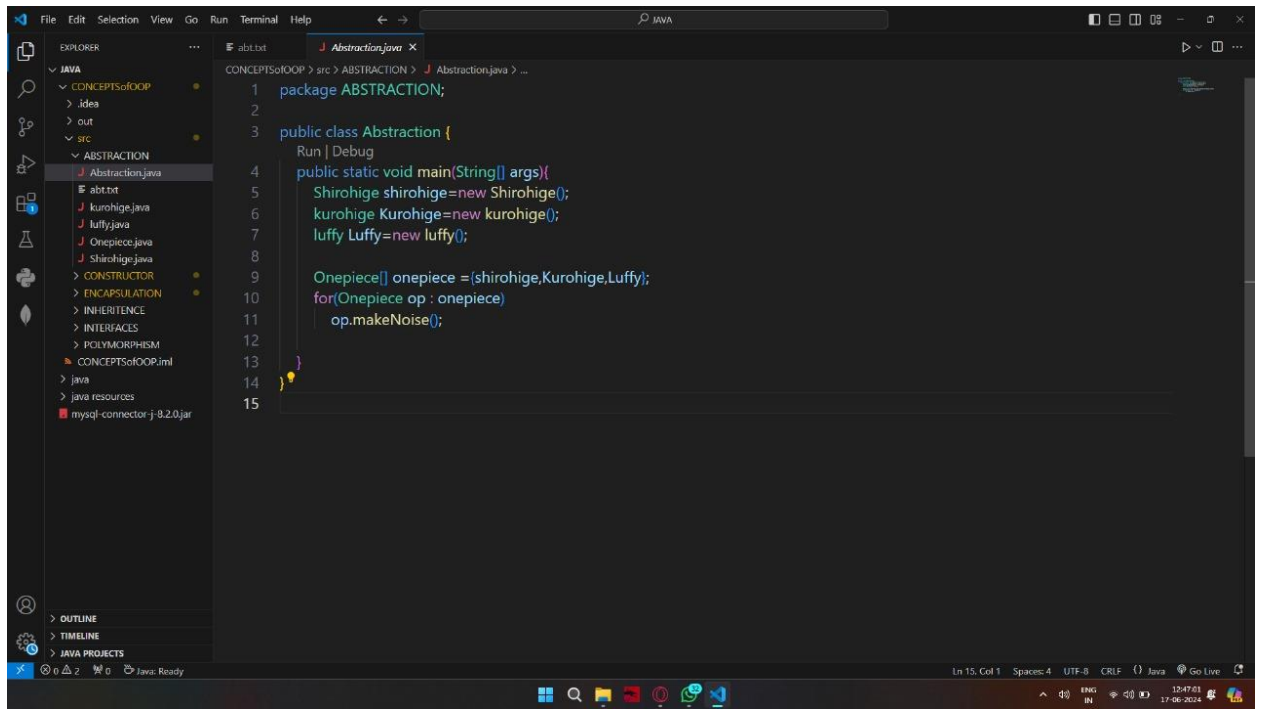
ABSTRACT CLASSES : if a class is abstract ,it cannot be instantiated. But it's subclasses can be instantiated

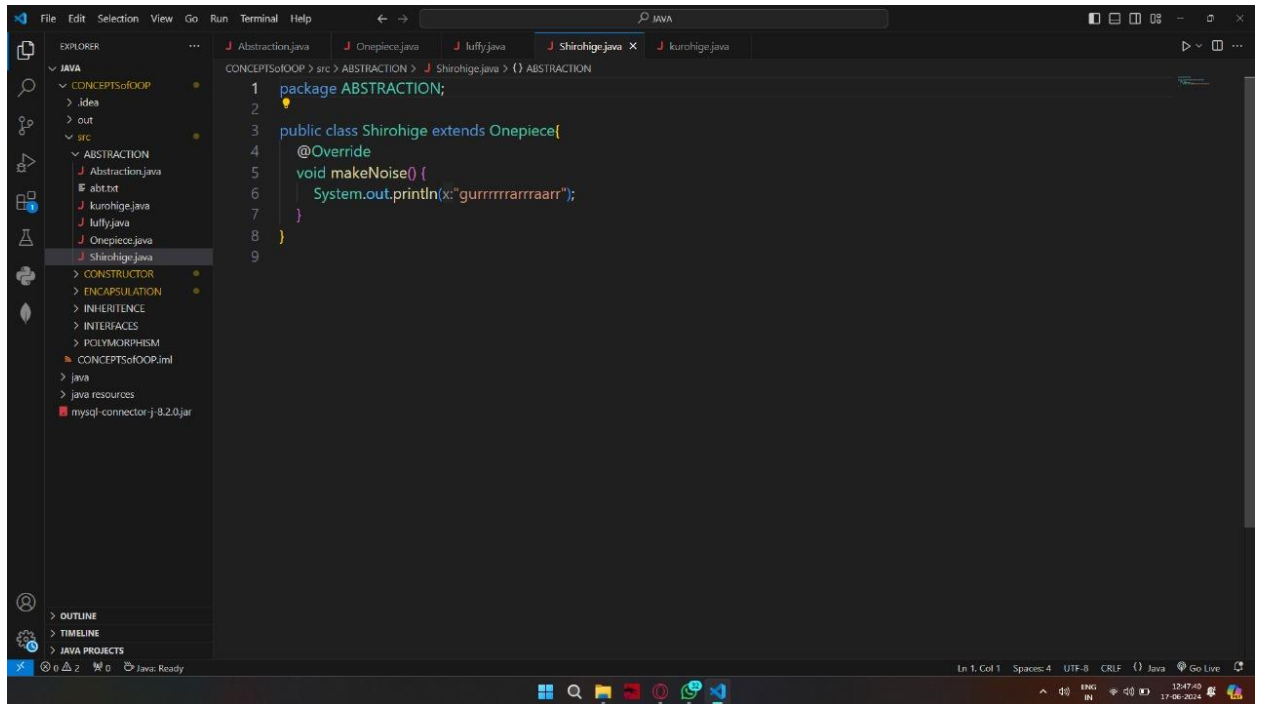
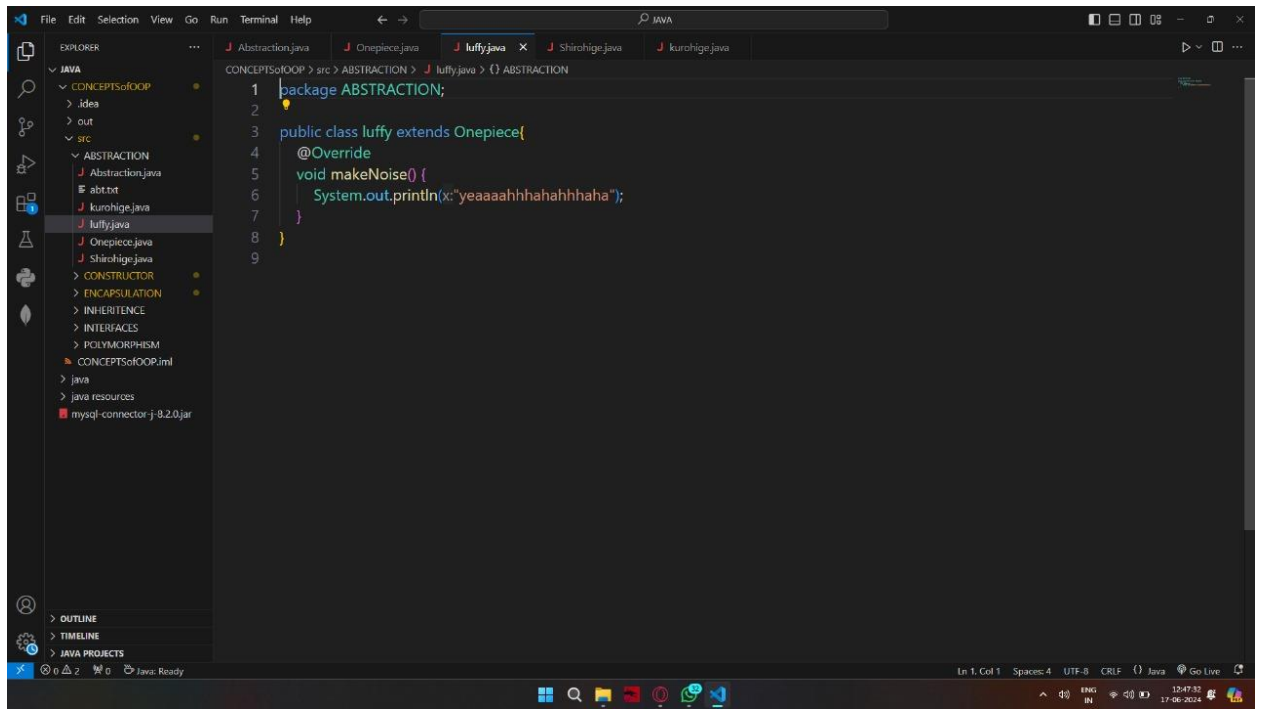
ABSTRACT METHOD : if method is abstract inside abstract class , then all the subclasses must have that method(override).

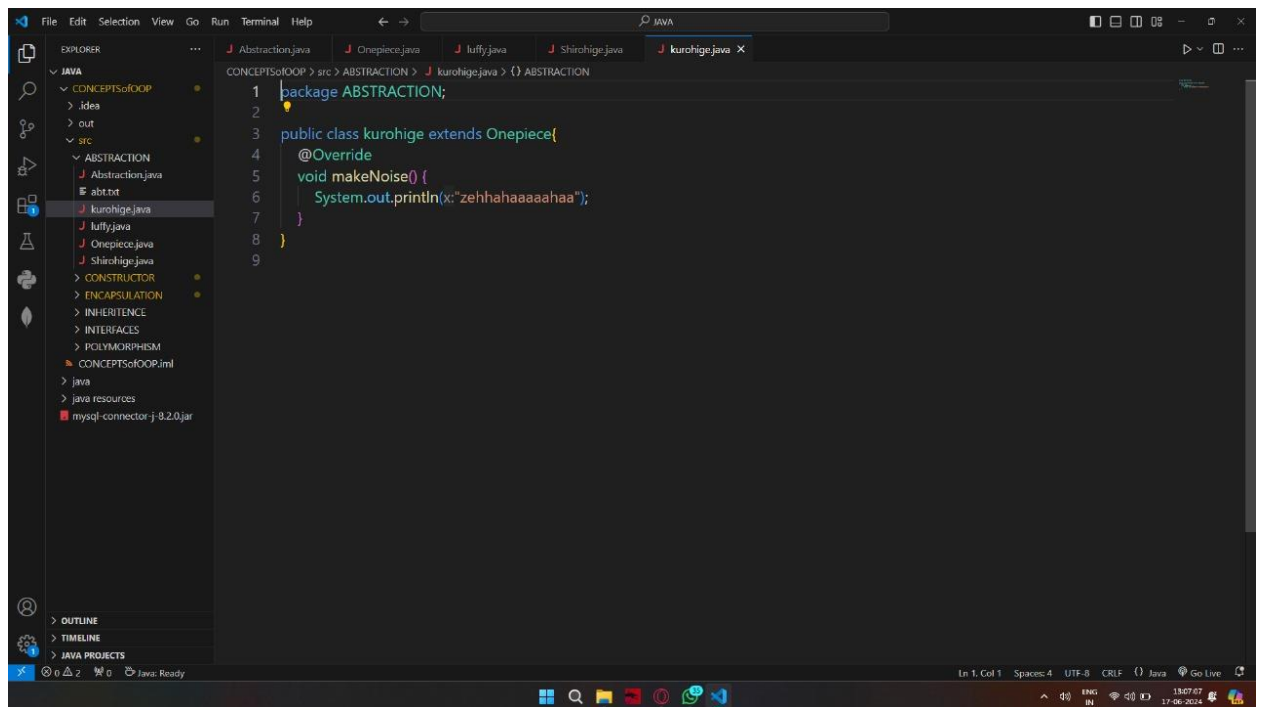
- < abstract void makeNoise(); > this is how abstract method is declared i.e no use of curly braces.

-the whole point of 'abstract' keyword is that it adds a layer of security to our program and this can be applied to both classes and methods

Ex:







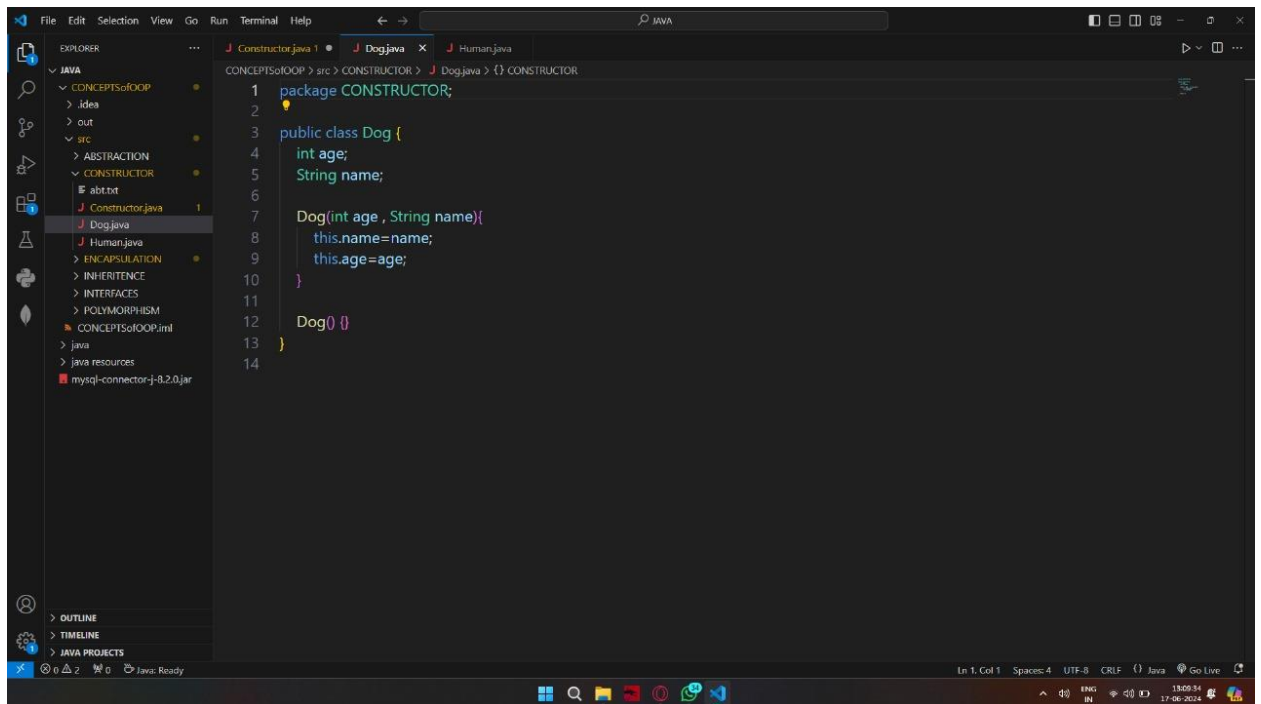
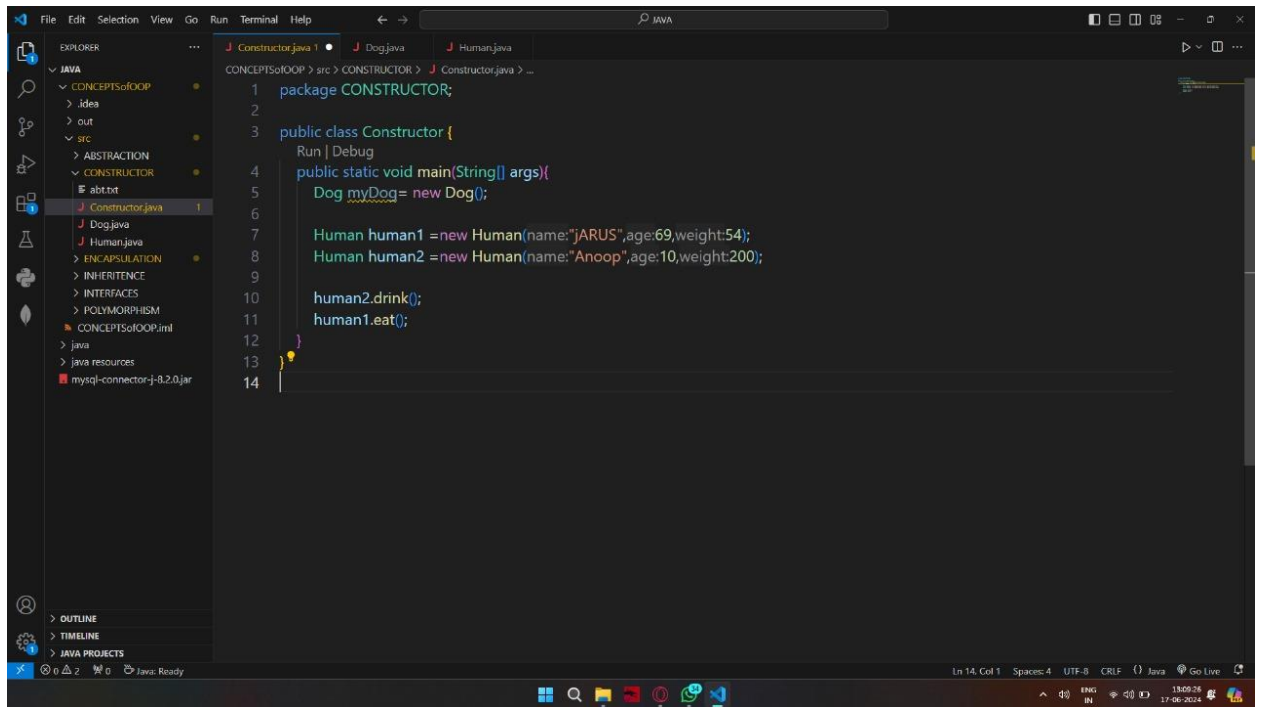
Java Constructors:

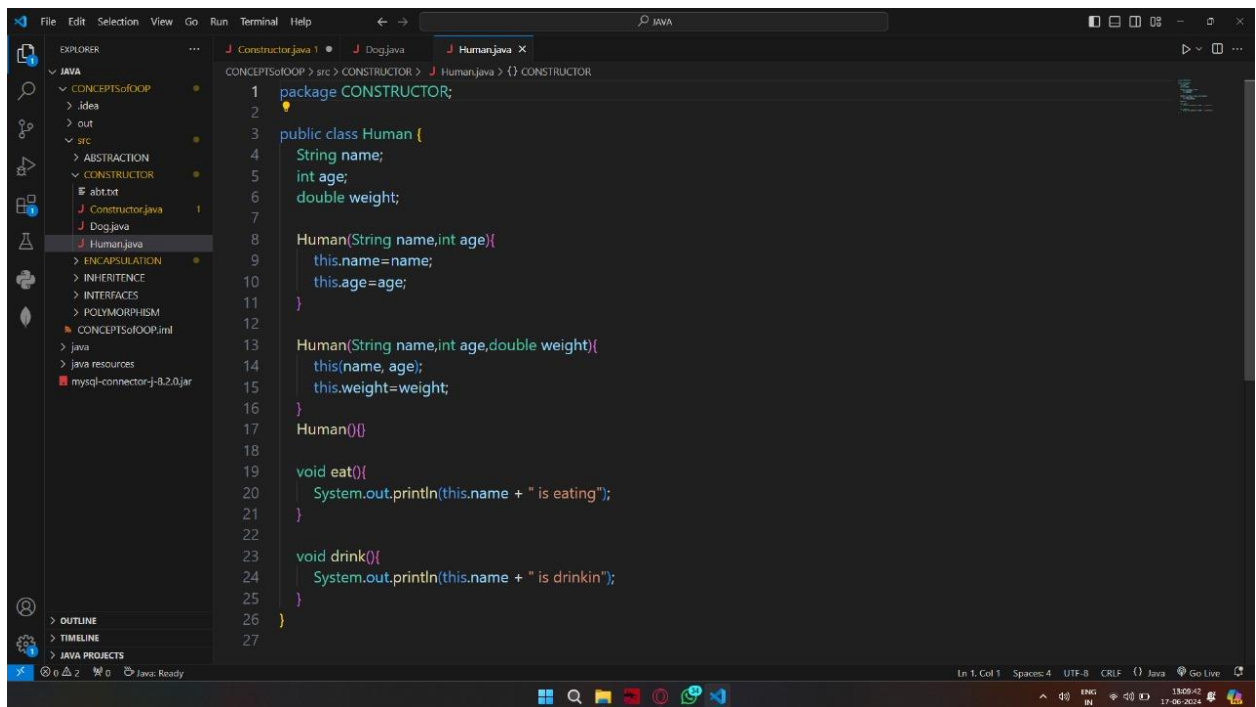
a special that is called when an obj is instantiated (created)

-Constructors don't have return type

-if constructor is private it cannot be accessed outside the class , i.e to restrict object creation

Ex:

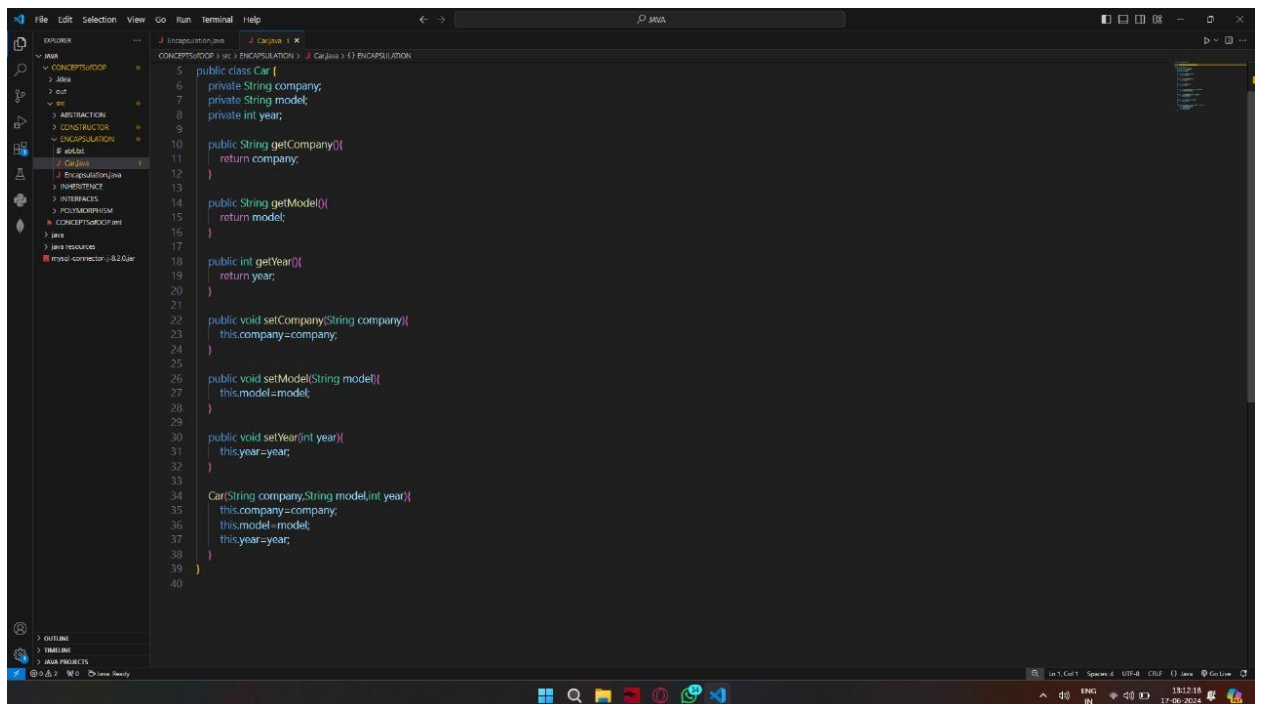
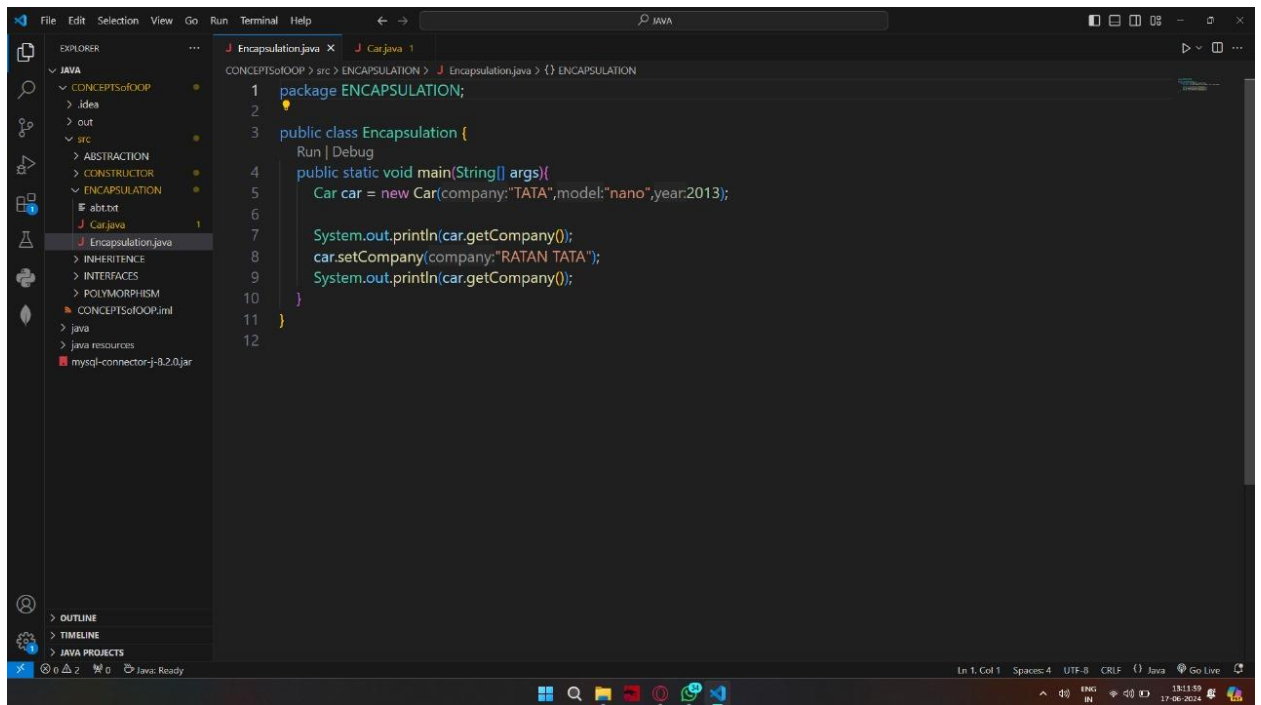




Java Encapsulation:

- attributes of class will be hidden or private,
- can be accessed only through getter and setter methods.
- You should make attributes private if you don't have reason to make it public/protected.

Ex:



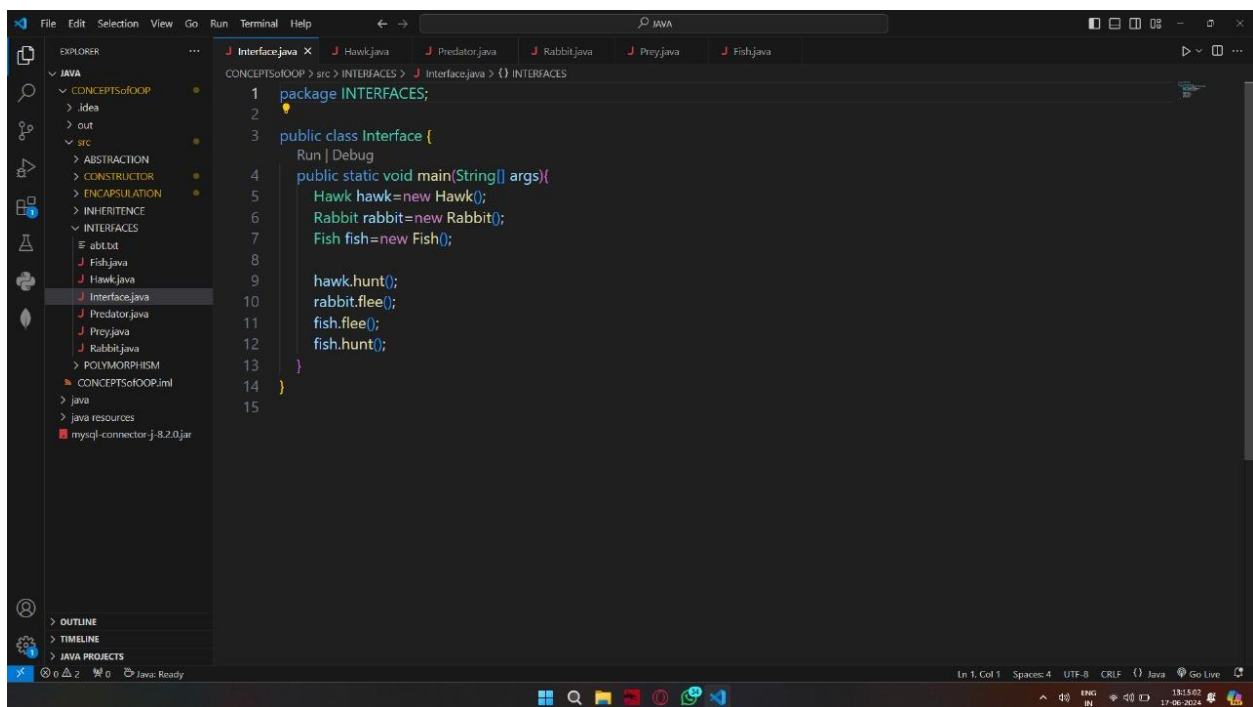
Java Interface:

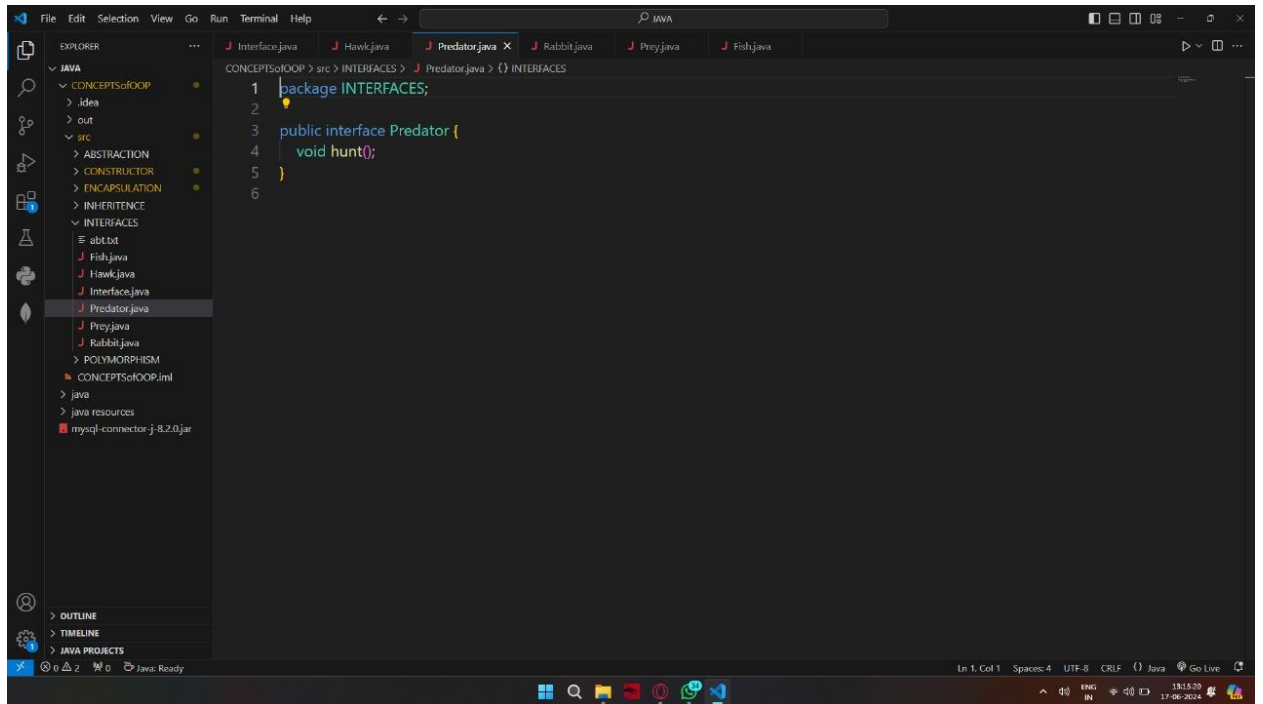
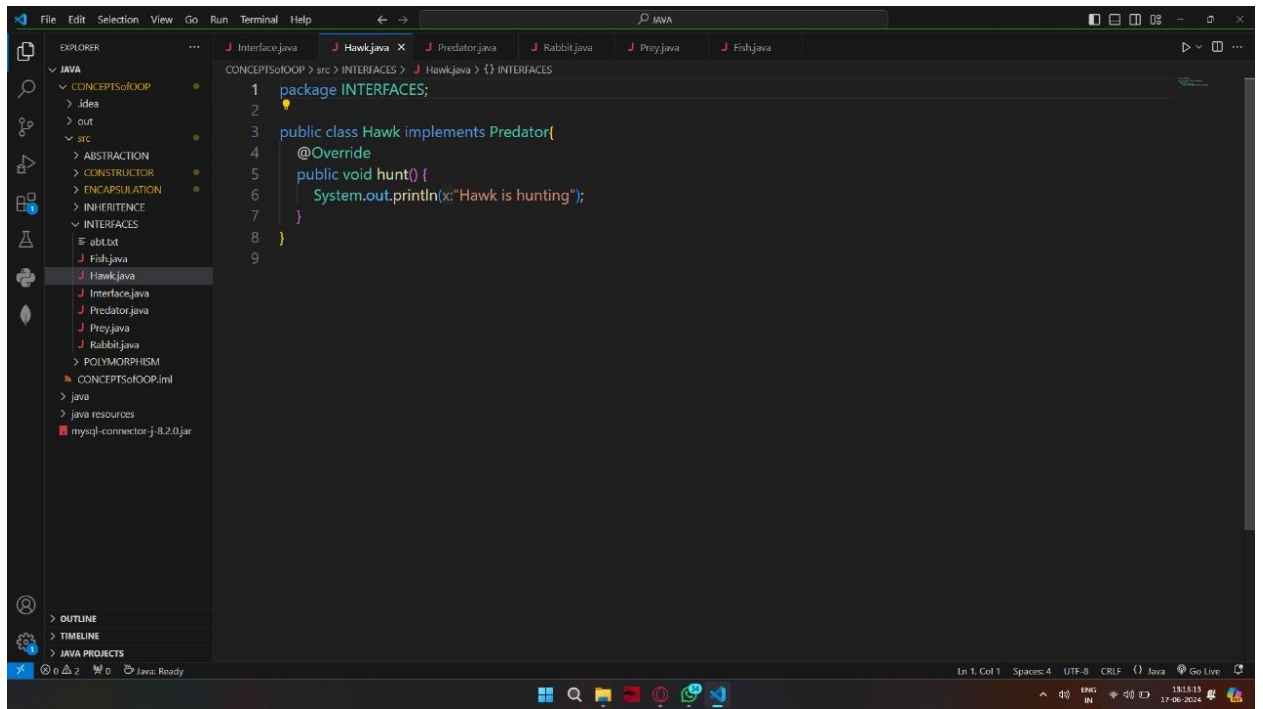
a template that can be applied to a class

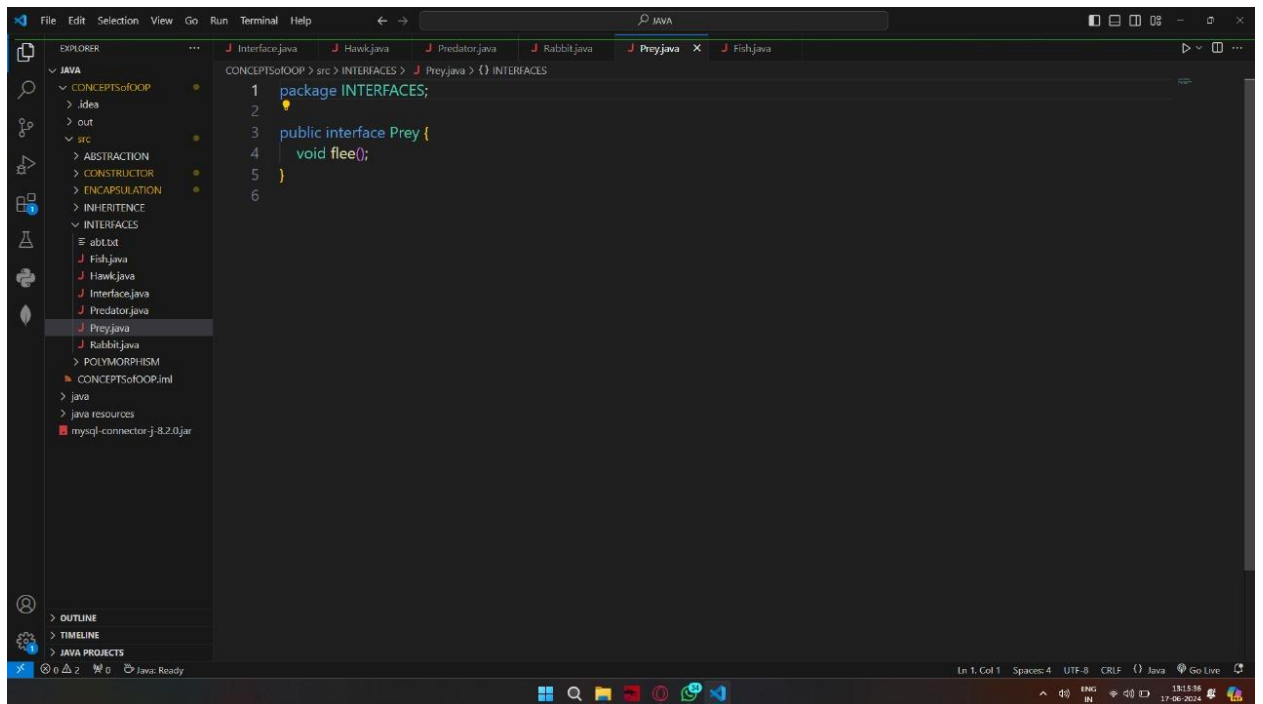
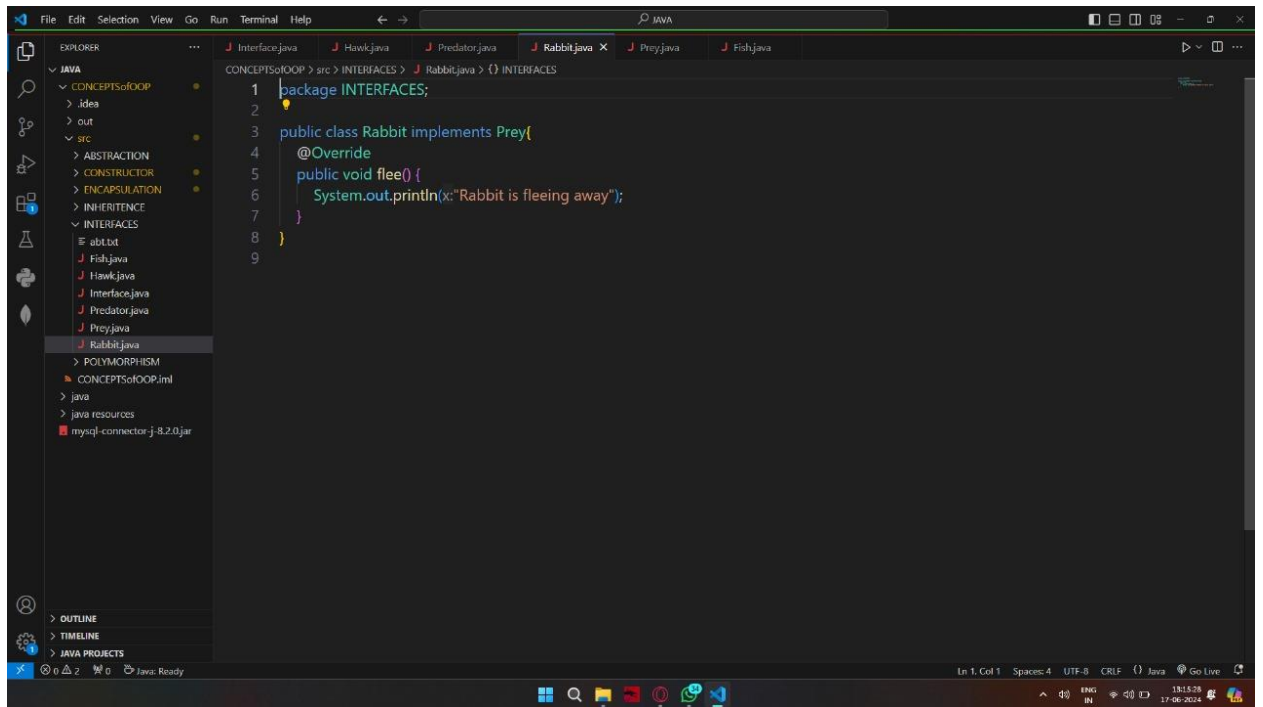
similar to inheritance ,but specifies what a class has/must do.

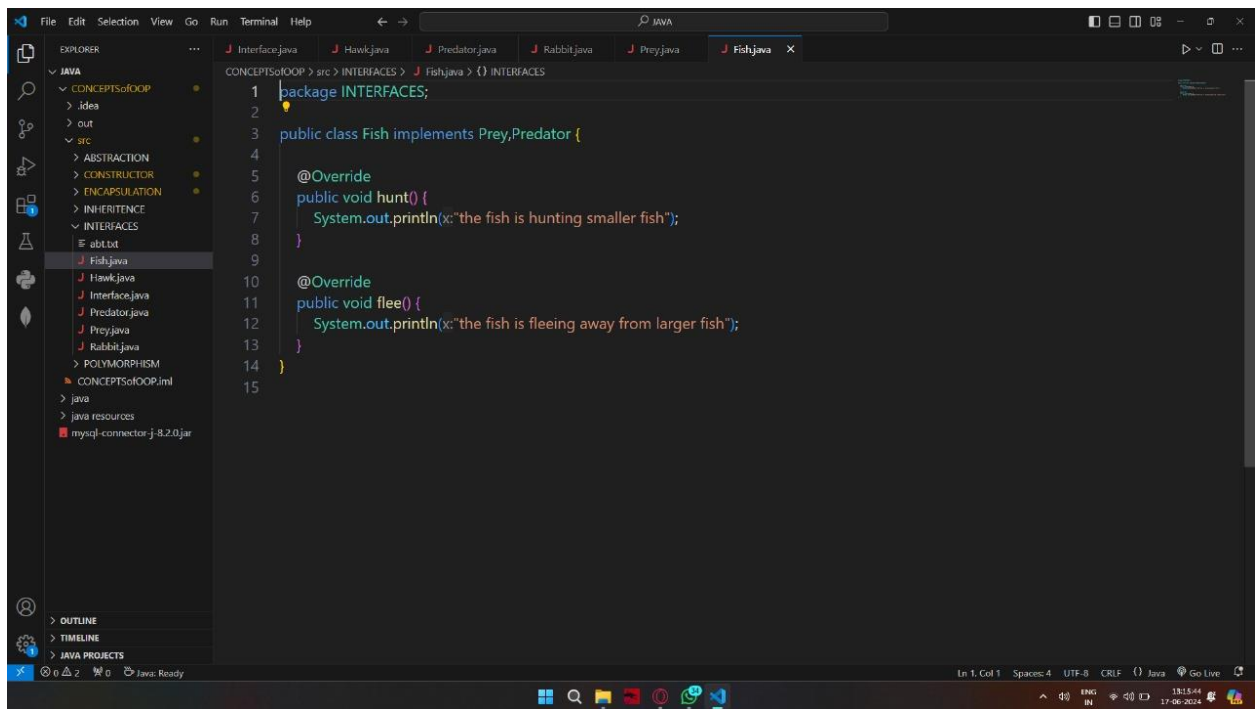
-classes can have one or more interface but only one inheritance

Ex:









Exception Handling:

Hierarchy of Exception Classes:

- Java's exception classes are organized in a hierarchy where Throwable is the root class.
- Direct subclasses include Error (serious errors not usually handled by applications) and Exception (base class for checked exceptions).

Types of Exceptions:

- Checked Exceptions: These are checked at compile-time. Examples include IOException, SQLException.
- Unchecked Exceptions (Runtime Exceptions): These are not checked at compile-time. Examples include NullPointerException, ArrayIndexOutOfBoundsException.

try-catch-finally:

- try: Used to enclose the code that might throw an exception.

-catch: Used to handle the exception if it occurs, specifying the type of exception to catch.

-finally: Optional block that follows a try-catch block; it executes whether an exception is thrown or not, typically used for cleanup.

throws:

-Used in method signatures to indicate that the method may throw one or more exceptions. It delegates the responsibility of handling exceptions to the caller method.

Multiple Catch Blocks:

-Allows catching different types of exceptions in separate catch blocks. Java allows multiple catch blocks to handle different exceptions that can be thrown by the try block.

Exception Handling with Method Overriding:

-Subclasses can override methods from superclasses even if those methods declare checked exceptions.

-Subclasses can choose to declare the same exception type, a subclass of the exception type, or no exception at all (unchecked exception).

Java Collection Framework:

Hierarchy of Collection Framework:

-Interfaces: Collection (extends Iterable) and Map.

-Classes: AbstractCollection, AbstractList, AbstractSet (provide skeletal implementations).

-Implementations: List (e.g., ArrayList, LinkedList), Set (e.g., HashSet, TreeSet), Queue (e.g., PriorityQueue), Map (e.g., HashMap, TreeMap).

Collection Interface:

- Root interface of the Collection framework.
- Contains basic operations like add, remove, contains, size, etc.

Iterator Interface:

- Allows iterating over elements in a collection.
- Methods include hasNext() and next().

Set, List, Queue, Map Interfaces:

- Set: Represents a collection that does not allow duplicate elements (e.g., HashSet, TreeSet).
- List: Ordered collection (e.g., ArrayList, LinkedList).
- Queue: Collection used for holding elements prior to processing (e.g., PriorityQueue).
- Map: Key-value pair collection (e.g., HashMap, TreeMap).

Comparator, Comparable Interfaces:

- Comparable: Interface used for natural ordering of objects (compareTo method).
- Comparator: Interface for defining custom ordering (compare method).

Classes:

- ArrayList: Implements dynamic arrays.
- Vector: Synchronized version of ArrayList.
- LinkedList: Doubly linked list implementation.
- PriorityQueue: Implements a priority queue.
- HashSet, LinkedHashSet, TreeSet: Implementations of Set interface with different characteristics.

-HashMap, ConcurrentHashMap: Implementations of Map interface, with ConcurrentHashMap supporting concurrent access.

Multithreading:

1. Lifecycle of a Thread :

- New : Created but not started.
- Runnable : Ready to run; 'start()' called.
- Running : 'run()' method executing.
- Blocked : Waiting for lock or I/O.
- Dead : Finished 'run()' or terminated.

2. Thread Priority :

- Ranges 1 to 10; higher means higher priority.
- Influences thread scheduling but OS-dependent.

3. Runnable Interface :

- Describes task for a thread ('run()' method).
- Used with 'Thread' or 'ExecutorService'.

4. start() Function :

- Starts thread execution by calling 'run()'.

5. Thread.sleep() Method :

- Pauses current thread for specified milliseconds.
- Can throw 'InterruptedException'.

6. Thread.run() in Java :

- Contains code executed by thread.
- Invoked when 'start()' starts a thread.

7. Deadlock :

- Threads wait indefinitely for each other's resources.
- Avoided with proper synchronization.

8. Synchronization :

- Controls access to shared resources.
- Prevents thread interference.

9. Method Level Lock :

- Synchronizes entire method execution.
- Ensures thread safety for instance methods.

10. Block Level Lock :

- Synchronizes specific code blocks.
- Allows finer-grained control over synchronization.

Executor Framework Java

1. Executor Framework :

- Manages thread execution with thread pools.
- Provides higher-level abstraction than managing threads directly.

2. Callable Interface :

- Similar to 'Runnable ' but can return a result and throw checked exceptions.
- Used with 'ExecutorService ' for asynchronous task execution.

Certainly! Here are two points each for the specified Java 8 features:

Java 8 Features

1. Functional Interfaces :

- Interfaces that have exactly one abstract method.
- Used for lambda expressions and method references.

2. Optional :

- Represents an object that may or may not contain a non-null value.
- Helps avoid 'NullPointerException ' by providing methods like 'isPresent() ', 'orElse() ', 'orElseGet() ', etc.

3. Default Methods :

- Methods in interfaces that have a default implementation.
- Allows interfaces to have methods with implementations without affecting implementing classes.

4. Stream API :

- Enables functional-style operations on sequences of elements.
- Supports operations such as 'filter ', 'map ', 'reduce ', 'collect '.
- Facilitates parallel execution of operations with 'parallelStream() '.

5. Java Time API :

- Introduces classes like 'LocalDate ', 'LocalTime ', 'LocalDateTime ', 'ZonedDateTime ', etc., for date and time handling.

- Provides immutable date-time objects with enhanced features over the legacy 'Date ' and 'Calendar ' classes.

6. Lambda Expressions :

- Anonymous functions that allow you to treat functionality as a method argument.
- Simplifies code by providing a concise syntax.

7. Method Reference :

- Provides a way to refer to methods or constructors without invoking them.
- Enhances the readability of lambda expressions.
- Types of method references include 'Static method ', 'Instance method of a particular object ', and 'Constructor '.

8. Metaspace :

- Replaces the 'Permanent Generation ' memory space in Java 8 and later versions.
- Stores metadata about classes and methods.

Core Spring Framework Annotations:

1. Component Scanning and Bean Definition:

- @Component: Indicates a class as a Spring component.
- @Repository: Specialization of @Component for data access classes.
- @Service: Specialization of @Component for service layer classes.
- @Controller: Specialization of @Component for Spring MVC controllers.
- @Configuration: Indicates that a class declares one or more @Bean methods.

2. Bean Lifecycle and Scope:

- `@Scope`: Defines the scope of a bean (singleton, prototype, etc.).
- `@PostConstruct`: Method annotated with this is executed after dependency injection and before the bean is put into service.
- `@PreDestroy`: Method annotated with this is executed just before the bean is removed from the container.

3. Dependency Injection:

- `@Autowired`: Enables automatic dependency injection.
- `@Qualifier`: Specifies which bean should be injected when multiple beans of the same type exist.
- `@Primary`: Indicates a primary bean when multiple candidates are available for autowiring.
- `@Inject`: Alternative to `@Autowired`, provided by the JSR-330 (Java Dependency Injection standard).

4. Aspect-Oriented Programming (AOP):

- `@Aspect`: Marks a class as an aspect, which is a modularization of a cross-cutting concern.
- `@Before`, `@After`, `@Around`: Advice annotations used to define aspects.

5. Transactional Management:

- `@Transactional`: Defines the scope of a single transaction method or class.

6. Testing:

- `@RunWith(SpringRunner.class)`: Runs Spring tests using JUnit or other test frameworks.
- `@SpringBootTest`: Bootstraps the Spring ApplicationContext for integration tests.

- @MockBean: Mocks a bean when used in integration tests.

Spring Boot Annotations:

1. Application Configuration:

- @SpringBootApplication: Combines @Configuration, @EnableAutoConfiguration, and @ComponentScan.
- @EnableAutoConfiguration: Enables Spring Boot's auto-configuration mechanism.

2. Web Applications:

- @Controller: Marks a class as a Spring MVC controller.
- @RestController: Specialization of @Controller that simplifies RESTful web services.
- @RequestMapping, @GetMapping, @PostMapping, @PutMapping, @DeleteMapping: Maps HTTP requests to handler methods.

3. Data Access:

- @Entity: Marks a class as a JPA entity.
- @RepositoryRestResource: Exposes Spring Data repositories as REST endpoints.

4. Configuration Properties:

- @ConfigurationProperties: Binds and validates external configuration properties to a POJO.

5. Spring Boot Actuator:

- @Endpoint: Defines a custom actuator endpoint.
- @Actuator: Enables Spring Boot Actuator endpoints.

6. Testing:

- `@SpringBootTest`: Loads the `ApplicationContext` and can be used with `@WebMvcTest` or `@DataJpaTest`.
- `@DataJpaTest`: Configures a test slice for JPA tests.
- `@WebMvcTest`: Configures a test slice for Spring MVC tests.

7. Spring Boot DevTools:

- `@DevToolsTest`: Loads Spring Boot DevTools configuration for integration tests.

8. Security:

- `@EnableWebSecurity`: Enables Spring Security's web security support.

Servlets:

WORKING :

Client sends request of a page let's say , if its dynamic page(made at runtime) → that req is sent to containers containing of servlets

DEPLOYMENT DESCRIPTOR :

In our container, there will be deployment descriptor file which will know which servlet to be called for a specific req.

And the file name is `web.xml`.

CONTAINER :

Tomcat is a web container which contains servlets.

We can interact b/w servlets , i.e call one servlet from another servlet ,
there are 2 ways to do it :

→ Request Dispatcher

→ Redirect

We can send data from one server to another / one servlet to another servlet using

→Session Management

→Request Dispatcher

using req dispatcher.

JSP:

stands for Java Server Pages.

JSP makes things so simple compared to Servlets.

When jsp file is ran , it is converted to a java file , i.e converts to servlets because we can only run servlets in tomcat container , we cannot run jsp in tomcat.

4 ways to add java code in jsp file

→ directive : to import packages `<%@ %>` :

`@page , @include , @taglib`

→ declaration : to write functions outside service() `<%! %>`

→ scriptlet : inside service() code `<% %>`

→ expression : something to print directly `<%= %>`

JDBC:

JDBC is an API used to interact with database using sql queries.

- 1) Import the package. (java.sql.*)
- 2) a) Load the driver

- b) Register the driver
- 3) Establish the connection
- 4) Create the statement (prepared statement , callable statement).
- 5) Execute the Query
- 6) Process Request
- 7) Close the connection

Hibernate:

-Hibernate ORM is an object–relational mapping tool for the Java programming language. It provides a framework for mapping an object-oriented domain model to a relational database.

-if we want to maintain relationships b/w different tables , we can use annotations like OnetoOne , OnetoMany , ManytoOne , ManytoMany.

-Hibernate is not only famous for ORM , but it also provides caching.

One of the famous and most widely used is ehcache.