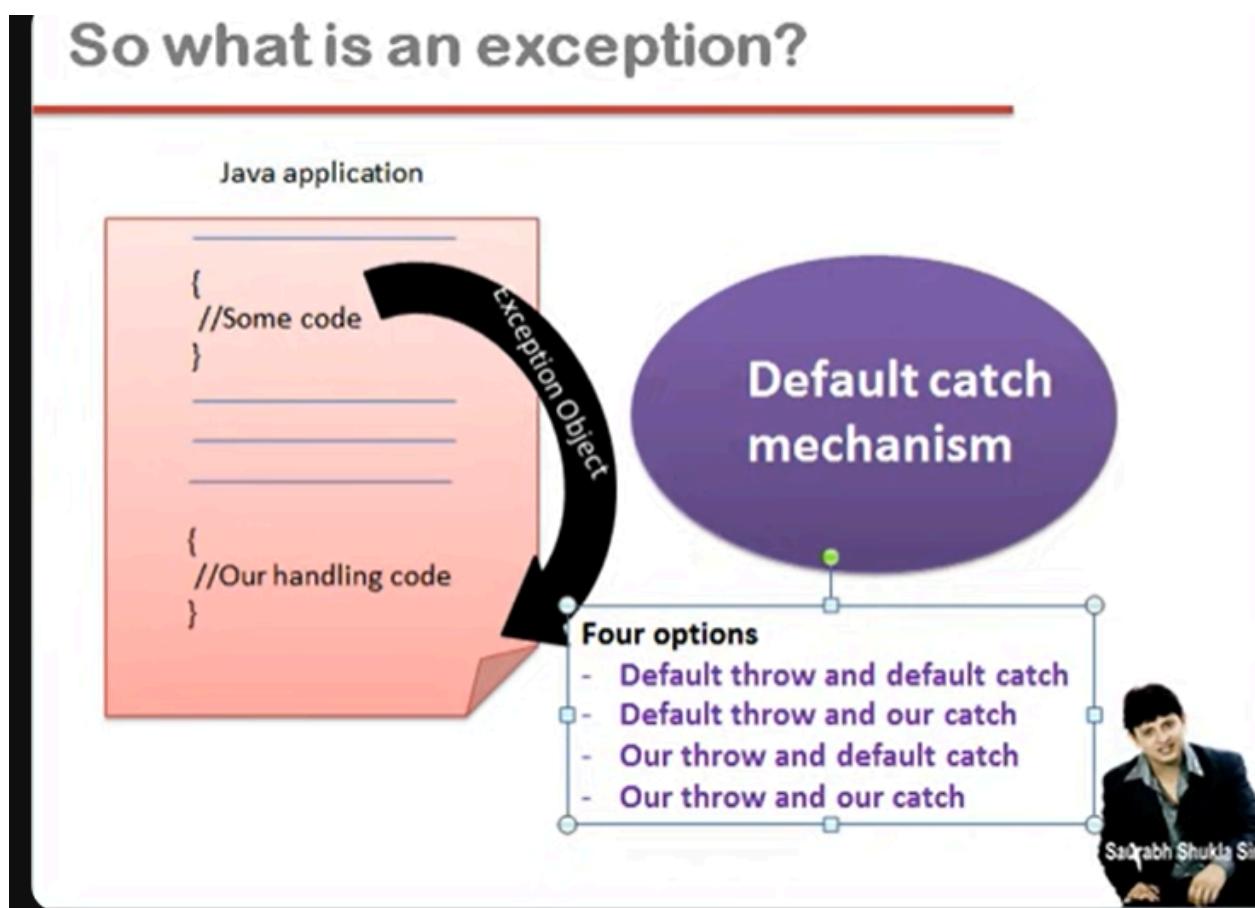
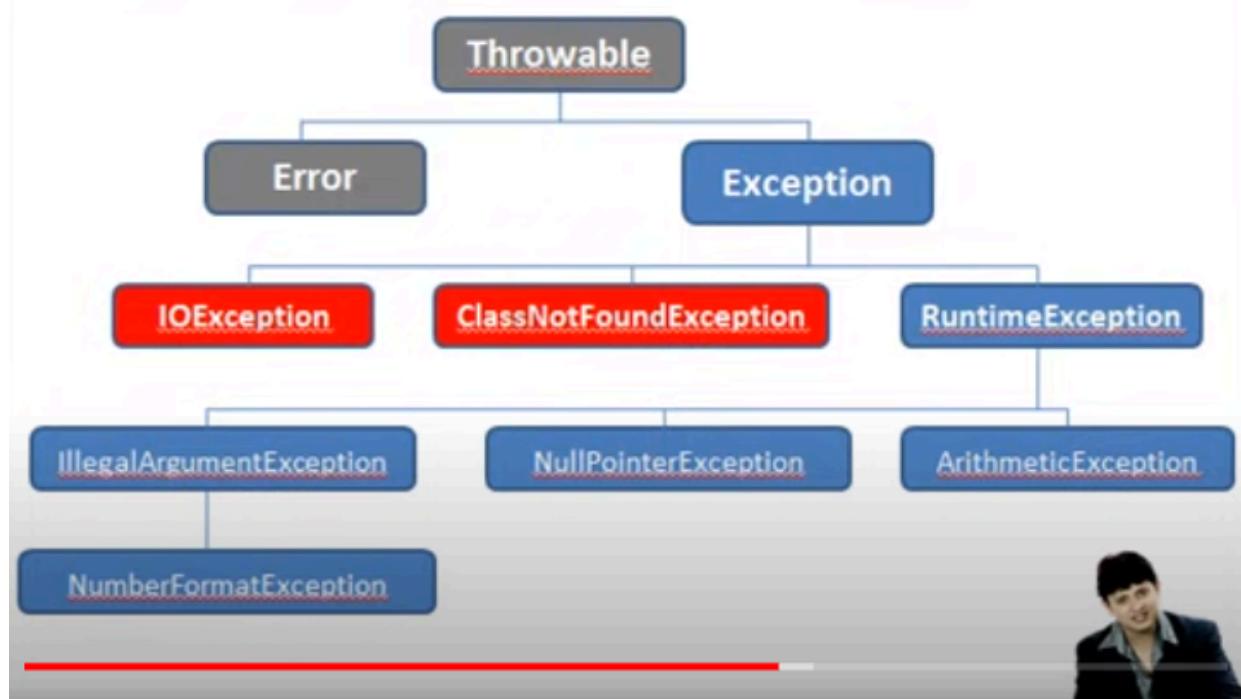


Exception

So what is an exception?



Class Hierarchy



A screenshot of an IDE showing the code for **Example.java**. The code defines a class **Example** with a **main** method. Inside the **main** method, there is a line of code that attempts to access the **length()** method of the variable **s1**, which is of type **String**. This line is highlighted in green.

```
1 class Example{  
2     public static void main(String[] args){  
3         String s1;  
4         System.out.println("First Line");  
5         System.out.println("String length is "+s1.length());  
6         System.out.println("Last Line");  
7     }  
8 }  
9
```

```
Example.java |  
Command Prompt  
G:\Java Programs>javac Example.java  
Example.java:5: error: variable s1 might not have been initialized  
    System.out.println("String length is "+s1.length());  
           ^  
1 error  
G:\Java Programs>javac Example.java  
G:\Java Programs>java Example  
First Line  
Exception in thread "main" java.lang.NullPointerException  
at Example.main(Example.java:5)  
G:\Java Programs>
```

Throwable

- The Throwable class provides a String variable that can be set by the subclasses to provide a detail message that provides more information of the exception occurred
- All classes of Throwables define a one-parameter constructor that takes a string as the detail message
- The class Throwable provides getMessage() function to retrieve an exception



Exceptions are of two types

- ❑ The class `Exception` represents exceptions that a program faces due to abnormal or special conditions during execution.
- ❑ Exceptions can be of 2 types: **Checked** (Compile time Exceptions)/ **Unchecked** (Run time Exceptions).

Unchecked Exceptions

- ❑ **Unchecked exceptions** are `RuntimeException` and any of its subclasses
- ❑ `ArrayIndexOutOfBoundsException`, `NullPointerException` and so on are all subclasses of the `java.lang.RuntimeException` class, which is a subclass of the `Exception` class.



```
1 class Example{
2     public static void main(String[] args){
3         try{
4             System.out.println(3/0);
5             System.out.println("In try");
6         }
7         catch(ArithmeticException e){
8             System.out.println("Exception: "+e.getMessage());
9         }
10        System.out.println("Hello");
11    }
12 }
13
```

Command Prompt

```
G:\Java Programs>javac Example.java
G:\Java Programs>java Example
Exception: / by zero
Hello
G:\Java Programs>
```

Try can have multiple catch statement

```
1 class Example{
2     public static void main(String[] args){
3         try{
4             System.out.println(3/0);
5             System.out.println("In try");
6         }
7         catch(NullPointerException e)
8         {
9             System.out.println("Exception: "+e.getMessage());
10        }
11        catch(ArithmeticException e){
12            System.out.println("Exception: "+e.getMessage());
13        }
14        System.out.println("Hello");
15    }
16 }
17 }
```

```
1 class Example{
2     public static void main(String[] args){
3         try{
4             System.out.println(3/0);
5             System.out.println("In try");
6         }
7         catch(NullPointerException e)
8         {
9             System.out.println("Exception: "+e.getMessage());
10        }
11        catch(ArithmeticException e){
12            System.out.println("Exception: "+e.getMessage());
13        }
14        System.out.println("Hello");
15    }
16 }
17 }
```

Remember

- For each try block there can be zero or more catch blocks, but only one finally block
- The catch blocks and finally block must always appear in conjunction with a try block
- A try block must be followed by either at least one catch block or one finally block.
- The order exception handlers in the catch block must be from the most specific exception



Four ways

- Default throw and default catch
- Default throw and our catch
- Our throw and default catch
- Our throw and our catch

Explicit throw

- ❑ A program can explicitly throw an exception using the throw statement besides the implicit exception thrown.

- ❑ Syntax:
 - `throw <throwableInstance>;`

Click to add title

- ❑ `throw <throwableInstance>;`

- ❑ The Exception reference must be of type Throwable class or one of its subclasses
- ❑ A detail message can be passed to the constructor when the exception object is created.



Humare project k hisab se wo error
H java k sidese ni

Our throw and default catch

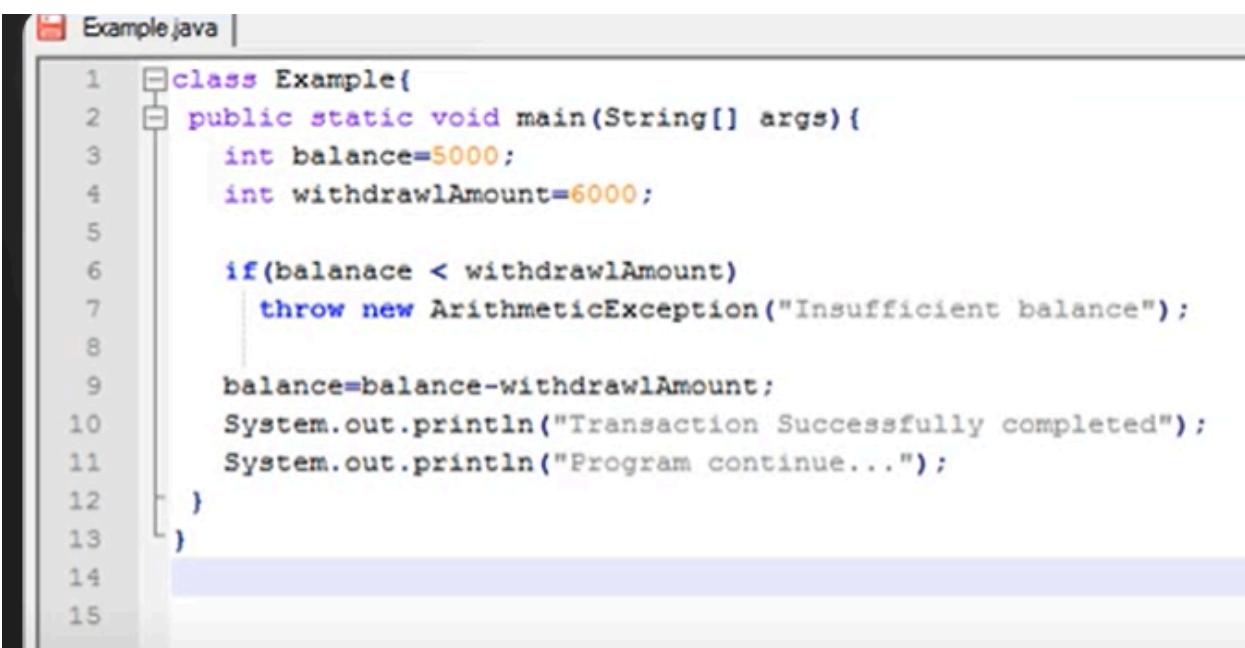
```
class Example{
    public static void main(String[] args){
        int balance=5000;
        int withdrawlAmount=6000;

        if(balanace < withdrawlAmount)
            throw new ArithmeticException("Insufficient balance");

        balance=balance-withdrawlAmount;
        System.out.println("Transaction Successfully completed");
        System.out.println("Program continue...");
    }
}
```



Sachin Shukla



A screenshot of a Java code editor showing the contents of a file named Example.java. The code is identical to the one shown in the previous slide, with line numbers 1 through 15 visible on the left. The code defines a class Example with a main method that checks if the balance is less than the withdrawal amount. If it is, it throws an ArithmeticException. Otherwise, it prints a success message and a continuation message.

We are writing arithmeticException because it is an object of the exception class.

Example.java

Command Prompt

```
G:\Java Programs>javac Example.java
Example.java:6: error: cannot find symbol
      if(balanace < withdrawlAmount)
      ^
symbol:   variable balanace
location: class Example
1 error

G:\Java Programs>javac Example.java

G:\Java Programs>java Example
Exception in thread "main" java.lang.ArithmaticException: Insufficient balance
at Example.main(Example.java:7)

G:\Java Programs>
```

Example.java

```
1  class Example{
2  public static void main(String[] args){
3      int balance=5000;
4      int withdrawlAmount=6000;
5      try
6      {
7          if(balance < withdrawlAmount)
8              throw new ArithmaticException("Insufficient balance");
9          balance=balance-withdrawlAmount;
10         System.out.println("Transaction Successfully completed");
11     }
12     catch(ArithmaticException e)
13     {
14         System.out.println("Exception: "+e.getMessage());
15     }
16     System.out.println("Program continue...");
```

One question

- Why should we throw an exception object?
 - Because we want to set a different message
 - Because java cannot recognize exceptional situation of business logic

Checked exception:

Compile Time Error in checked exception

- Checked Exceptions forces programmers to deal with the exception that may be thrown
- IOException, SQLException, IllegalThreadStateException, etc are checked exceptions
- "checked" means they will be checked at compile time itself

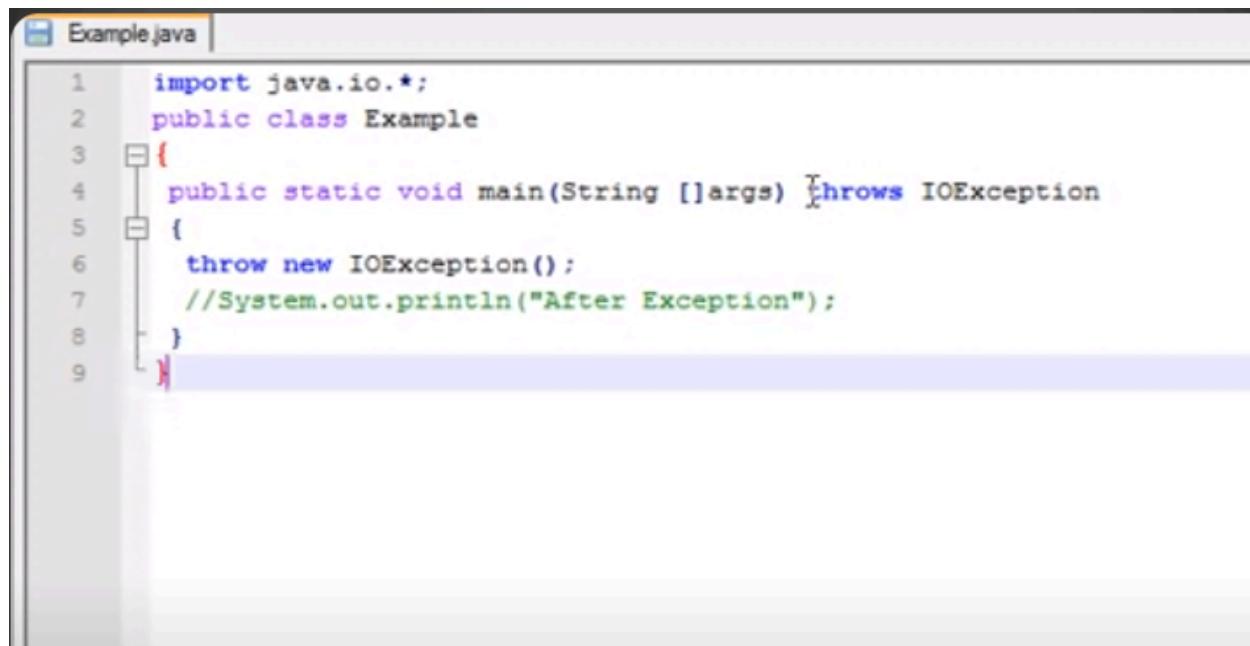
throws

- ❑ A **throws** clause can be used in the method prototype

Method() throws <ExceptionType1>, ...,
<ExceptionTypeN>

{

}



The screenshot shows a Java code editor window titled "Example.java". The code is as follows:

```
1 import java.io.*;
2 public class Example
3 {
4     public static void main(String []args) throws IOException
5     {
6         throw new IOException();
7         //System.out.println("After Exception");
8     }
9 }
```

The word "throws" is highlighted in blue, and the exception type "IOException" is also highlighted in blue. The code editor has a light gray background with dark gray horizontal and vertical scroll bars.

```
1 import java.io.*;
2 public class Example
3 {
4     public static void main(String []args) throws IOException
5     {
6         throw new IOException();
7         //System.out.println("After Exception");
8     }
9 }
```

Ya to hume try catch laga k thik karna h exception ya java ko bolna h thik karne throws laga k

```
1 import java.io.*;
2 public class Example
3 {
4     public static void main(String []args)
5     {
6         ...
7         try
8         {
9             throw new IOException();
10            //System.out.println("After Exception");
11        }
12        catch(IOException e)
13        {
14            System.out.println("Exception:"+e.getMessage());
15        }
16    }
17 }
```

throws

- ❑ The throws keyword in java programming language is applicable to a method to indicate that the method raises particular type of exception while being processed.
- ❑ The throws keyword in java programming language takes arguments as a list of the objects of type `java.lang.Throwable`s class.

<https://www.geeksforgeeks.org/throw-throws-java/>

The 'throw' keyword is used to explicitly throw an exception from within a block of code or a method. The 'throws' keyword is used in the method signature to declare the exceptions that a method can potentially throw. The 'throw' keyword can only throw a single exception at a time

String

```
class Main {  
    public static void main(String[] args) {  
  
        // create a string  
        String greet = "Hello! World";  
        System.out.println("String: " + greet);  
  
        // get the length of greet  
        int length = greet.length();  
        System.out.println("Length: " + length);  
    }  
}
```

```
}
```

Int

```
class Main {  
    public static void main(String[] args) {  
  
        int range = -4250000;  
        System.out.println(range); // print -4250000  
    }  
}
```

—

```
class Main {  
    public static void main(String[] args) {  
  
        // create first string  
        String first = "Java ";  
        System.out.println("First String: " + first);  
  
        // create second  
        String second = "Programming";  
        System.out.println("Second String: " + second);  
  
        // join two strings  
        String joinedString = first.concat(second);  
        System.out.println("Joined String: " + joinedString);  
    }  
}
```

—

```
// include double quote  
String example = "This is the "String" class";
```

```
// use the escape character  
String example = "This is the \"String\\\" class.";
```

In Java, strings are immutable. This means, once we create a string, we cannot change that string.

Since strings in Java are objects, we can create strings using the `new` keyword as well. For example,

```
// create a string using the new keyword
String name = new String("Java String");
```

In the above example, we have created a string `name` using the `new` keyword.

Here, when we create a string object, the `String()` constructor is invoked.

Create Java Strings using the new keyword

```
class Main {
    public static void main(String[] args) {

        // create a string using new
        String name = new String("Java String");

        System.out.println(name); // print Java String
    }
}
```

how to take input a string in java

In Java, you can take input as a string from the user using the **Scanner class, which is part of the `java.util` package**. Here's a simple example:

```
import java.util.Scanner;
```

```
public class StringInputExample {  
    public static void main(String[] args) {  
        // Create a Scanner object to read input from the user  
        Scanner scanner = new Scanner(System.in);  
  
        // Prompt the user to enter a string  
        System.out.print("Enter a string: ");  
  
        // Read the entered string  
        String userInput = scanner.nextLine();  
  
        // Display the entered string  
        System.out.println("You entered: " + userInput);  
  
        // Close the scanner to release resources  
        scanner.close();  
    }  
}
```

Remember to close the Scanner object after you're done using it to free up system resources. The `scanner.close();` statement is used for this purpose.

Implicit = automatically
Explicit = forcefully

Java Constructors

- Constructors are invoked implicitly when you instantiate objects.

- The two rules for creating a constructor are:
 - The name of the constructor should be the same as the class.
 - A Java constructor must not have a return type.
- If a class doesn't have a constructor, the Java compiler automatically creates a default constructor during run-time. The default constructor initializes instance variables with default values. For example, the `int` variable will be initialized to `0`
- Constructor types:
 - No-Arg Constructor - a constructor that does not accept any arguments
 - Parameterized constructor - a constructor that accepts arguments
 - Default Constructor - a constructor that is automatically created by the Java compiler if it is not explicitly defined.
- A constructor cannot be `abstract` or `static` or `final`.
- A constructor can be overloaded but can not be overridden

Modifier	Description
Default	declarations are visible only within the package (package private)
Private	declarations are visible within the class only
Protected	declarations are visible within the package or all subclasses
Public	declarations are visible everywhere

Note: We cannot declare classes and interfaces private in Java. However, the nested classes can be declared private. To learn more, visit [Java Nested and Inner Class](#).

==

We cannot declare classes or interfaces `protected` in Java.

==

The protected access specifier in Java **allows members to be accessed within the same class, subclasses, and classes in the same package**. This means that protected members can be accessed by the class itself, its subclasses (even if they are in a different package), and other classes in the same package.

	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 non-subclass	no	no	no	yes

this Keyword

In Java, this keyword is used to refer to the current object inside a method or a constructor. For example,

```

class Main {
    int instVar;

    Main(int instVar){
        this.instVar = instVar;
        System.out.println("this reference = " + this);
    }

    public static void main(String[] args) {
        Main obj = new Main(8);
        System.out.println("object reference = " + obj);
    }
}

```

In Java, this keyword is used to refer to the current object inside a method or a constructor.

In Java, the `final` keyword is used to denote constants. It can be used with variables, methods, and classes.

Once any entity (variable, method or class) is declared `final`, it can be assigned only once. That is,

- the final variable cannot be reinitialized with another value
- the final method cannot be overridden
- the final class cannot be extended

The `instanceof` operator in Java is used to check whether an object is an instance of a particular class or not.

Its syntax is

```
objectName instanceof className;
```

Here, if `objectName` is an instance of `className`, the operator returns `true`.

Otherwise, it returns `false`.

=====

We have used the `@Override` annotation to tell the compiler that we are overriding a method. However, the annotation is not mandatory. To learn more, visit [Java Annotations](#).

==

super Keyword in Java Inheritance

Previously we saw that the same method in the subclass overrides the method in superclass.

In such a situation, the `super` keyword is used to call the method of the parent class from the method of the child class.

```
class Animal {
```

```
    // method in the superclass
    public void eat() {
        System.out.println("I can eat");
    }
}
```

```
// Dog inherits Animal
class Dog extends Animal {
```

```

// overriding the eat() method
@Override
public void eat() {

    // call method of superclass
    super.eat();
    System.out.println("I eat dog food");
}

// new method in subclass
public void bark() {
    System.out.println("I can bark");
}
}

class Main {
    public static void main(String[] args) {

        // create an object of the subclass
        Dog labrador = new Dog();

        // call the eat() method
        labrador.eat();
        labrador.bark();
    }
}

```

Output

```

I can eat
I eat dog food
I can bark

```

In the above example, the `eat()` method is present in both the base class `Animal` and the derived class `Dog`. Notice the statement,

```
super.eat();
```

Here, the `super` keyword is used to call the `eat()` method present in the superclass.

We can also use the `super` keyword to call the constructor of the superclass from the constructor of the subclass. To learn more, visit [Java super keyword](#).

- The most important use of inheritance in Java is code reusability. The code that is present in the parent class can be directly used by the child class.
- Method overriding is also known as runtime polymorphism. Hence, we can achieve Polymorphism in Java with the help of inheritance.

Note: Java doesn't support multiple inheritance. However, we can achieve multiple inheritance using interfaces. To learn more, visit [Java implements multiple inheritance](#).

Overriding

Notice the use of the `@Override` annotation in our example. In Java, annotations are the metadata that we used to provide information to the compiler. Here, the `@Override` annotation specifies the compiler that the method after this annotation overrides the method of the superclass.

It is not mandatory to use `@Override`. However, when we use this, the method should follow all the rules of overriding. Otherwise, the compiler will generate an error.

Java Overriding Rules

- Both the superclass and the subclass must have the same method name, the same return type and the same parameter list.
- We cannot override the method declared as `final` and `static`.
- We should always override abstract methods of the superclass (will be discussed in later tutorials).

=====

A common question that arises while performing overriding in Java is:

Can we access the method of the superclass after overriding?

Well, the answer is Yes. To access the method of the superclass from the subclass, we use the `super` keyword.

```
class Animal {  
    public void displayInfo() {  
        System.out.println("I am an animal.");  
    }  
}
```

```
class Dog extends Animal {  
    public void displayInfo() {
```

```
    super.displayInfo();
    System.out.println("I am a dog.");
}
}

class Main {
    public static void main(String[] args) {
        Dog d1 = new Dog();
        d1.displayInfo();
    }
}
```

Use Super Keyword-

```
class Animal {
    public void displayInfo() {
        System.out.println("I am an animal.");
    }
}
```

```
class Dog extends Animal {
    public void displayInfo() {
        super.displayInfo();
        System.out.println("I am a dog.");
    }
}
```

```
class Main {
    public static void main(String[] args) {
        Dog d1 = new Dog();
        d1.displayInfo();
    }
}
```

====

```
class Animal {  
    protected void displayInfo() {  
        System.out.println("I am an animal.");  
    }  
}  
  
class Dog extends Animal {  
    public void displayInfo() {  
        System.out.println("I am a dog.");  
    }  
}
```

```
class Main {  
    public static void main(String[] args) {  
        Animal d1 = new Dog();  
        d1.displayInfo();  
    }  
}
```

is this code is correct?

ChatGPT

ChatGPT

Yes, the code you provided is correct, and it demonstrates an example of method overriding in Java.

In the Animal class, there is a method called displayInfo() that prints "I am an animal." The Dog class extends Animal and overrides the displayInfo() method with its own implementation, which prints "I am a dog."

In the Main class, you create an instance of Dog and assign it to a variable of type Animal. This is an example of polymorphism. When you call d1.displayInfo(), it calls the overridden method in the Dog class, and the output will be "I am a dog."

So, the output of this code will be:

css

Copy code

I am a dog.

User

You

what is meant by instance of dog?

ChatGPT

ChatGPT

In the context of object-oriented programming, "**instance of Dog**" refers to an object that is created from the Dog class. In your provided code:

java

Copy code

```
Animal d1 = new Dog();
```

Here, new Dog() creates a new object of the Dog class, and it is assigned to a variable of type Animal. This is possible because of polymorphism and inheritance. The variable d1 is an instance of Animal, but it refers to an actual object of the Dog class.

In other words, d1 is an instance of the superclass Animal, but at runtime, it refers to an object that is an instance of the subclass Dog. This is an example of polymorphism, where a reference variable of a superclass type can refer to an object of a subclass type.

In the provided code, when you call d1.displayInfo(), it invokes the overridden method in the Dog class, demonstrating the dynamic polymorphic behavior.

Example 2: Use of super Keyword

```
class Animal {  
    public void displayInfo() {  
        System.out.println("I am an animal.");  
    }  
}
```

```
class Dog extends Animal {  
    public void displayInfo() {
```

```
    super.displayInfo();
    System.out.println("I am a dog.");
}
}

class Main {
    public static void main(String[] args) {
        Dog d1 = new Dog();
        d1.displayInfo();
    }
}
Run Code
```

Output:

I am an animal.
I am a dog.

It is important to note that constructors in Java are not inherited. Hence, there is no such thing as constructor overriding in Java.

However, we can call the constructor of the superclass from its subclasses. For that, we use `super()`. To learn more, visit [Java super keyword](#).

In Java, abstract classes are created to be the superclass of other classes. And, if a class contains an abstract method, it is mandatory to override it.

Super keyword

The `super` keyword in Java is used in subclasses to access superclass members (attributes, constructors and methods).

Uses of super keyword

1. To call methods of the superclass that is overridden in the subclass.
 2. To access attributes (fields) of the superclass if both superclass and subclass have attributes with the same name.
 3. To explicitly call superclass no-arg (default) or parameterized constructor from the subclass constructor.
3. Use of `super()` to access superclass constructor

As we know, when an object of a class is created, its default constructor is automatically called.

To explicitly call the superclass constructor from the subclass constructor, we use `super()`. It's a special form of the `super` keyword.

`super()` can be used only inside the subclass constructor and must be the first statement.\\

However, using `super()` is not compulsory. Even if `super()` is not used in the subclass constructor, the compiler implicitly calls the default constructor of the superclass.

So, why use redundant code if the compiler automatically invokes super()?

It is required if the parameterized constructor (a constructor that takes arguments) of the superclass has to be called from the subclass constructor.

The parameterized `super()` must always be the first statement in the body of the constructor of the subclass, otherwise, we get a compilation error.

Example 5: Call Parameterized Constructor Using super()

```
class Animal {  
  
    // default or no-arg constructor  
  
    Animal() {  
  
        System.out.println("I am an animal");  
  
    }  
  
    // parameterized constructor  
  
    Animal(String type) {  
  
        System.out.println("Type: "+type);  
  
    }  
}
```

```
class Dog extends Animal {  
  
    // default constructor  
  
    Dog() {  
  
        // calling parameterized constructor of the superclass  
  
        super("Animal");  
  
        System.out.println("I am a dog");  
  
    }  
  
}  
  
class Main {  
  
    public static void main(String[] args) {  
  
        Dog dog1 = new Dog();  
  
    }  
  
}
```

Run Code

Output

```
Type: Animal
```

```
I am a dog
```

The compiler can automatically call the no-arg constructor. However, it cannot call parameterized constructors.

If a parameterized constructor has to be called, we need to explicitly define it in the subclass constructor.

Abstract class

The abstract class in Java cannot be instantiated (**we cannot create objects of abstract classes**). We use the `abstract` keyword to declare an abstract class. For example,

```
// create an abstract class
abstract class Language {
    // fields and methods
}
```

```
...
```



```
// try to create an object Language
// throws an error
Language obj = new Language();
```

An abstract class can have both the regular methods and abstract methods. For example,

```
abstract class Language {
```

```
// abstract method  
abstract void method1();  
  
// regular method  
void method2() {  
    System.out.println("This is regular method");  
}  
}
```

An abstract class can have both the regular methods and abstract methods. For example,

```
abstract class Language {  
  
    // abstract method  
    abstract void method1();  
  
    // regular method  
    void method2() {  
        System.out.println("This is regular method");  
    }  
}
```

Java Abstract Method

A method that doesn't have its body is known as an abstract method.

We use the same abstract keyword to create abstract methods. For example,

abstract void display();

Here, display() is an abstract method. The body of display() is replaced by ;.

If a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error. For example,

```
// error  
// class should be abstract  
class Language {
```

```
// abstract method  
abstract void method1();  
}
```

Key Points to Remember

- We use the `abstract` keyword to create abstract classes and methods.
- ***The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes).*** We use the `abstract` keyword to declare an abstract class.
- ***A method that doesn't have its body is known as an abstract method. We use the same abstract keyword to create abstract methods. `abstract void display();`***
- An abstract method doesn't have any implementation (method body).
- A class containing abstract methods should also be abstract.
- We cannot create objects of an abstract class.
- To implement features of an abstract class, we inherit subclasses from it and create objects of the subclass.
- A subclass must override all abstract methods of an abstract class.
However, if the subclass is declared abstract, it's not mandatory to override abstract methods.
- We can access the static attributes and methods of an abstract class using the reference of the abstract class. For example,

```
Animal.staticMethod();
```

An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).

We use the **interface** keyword to create an interface in Java. For example,

```
interface Language {  
    public void getType();  
  
    public void getVersion();  
}
```

Here,

Language is an interface.

It includes abstract methods: getType() and getVersion().

Implementing an Interface

Like abstract classes, we cannot create objects of interfaces.

To use an interface, other classes must implement it. We use the `implements` keyword to implement an interface.

```
interface Polygon {  
    void getArea(int length, int breadth);  
}  
  
// implement the Polygon interface  
class Rectangle implements Polygon {  
  
    // implementation of abstract method  
    public void getArea(int length, int breadth) {  
        System.out.println("The area of the rectangle is " + (length * breadth));  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Rectangle r1 = new Rectangle();  
        r1.getArea(5, 6);  
    }  
}
```

```
The area of the rectangle is 30
```

In the above example, we have created an interface named `Polygon`. The interface contains an abstract method `getArea()`.

Here, the `Rectangle` class implements `Polygon`. And, provides the implementation of the `getArea()` method.

```
// create an interface
interface Language {
    void getName(String name);
}

// class implements interface
class ProgrammingLanguage implements Language {

    // implementation of abstract method
    public void getName(String name) {
        System.out.println("Programming Language: " + name);
    }
}

class Main {
    public static void main(String[] args) {
        ProgrammingLanguage language = new ProgrammingLanguage();
        language.getName("Java");
    }
}
```

Similar to classes, interfaces can extend other interfaces. The `extends` keyword is used for extending interfaces. For example,

```
interface Line {
    // members of Line interface
}

// extending interface
interface Polygon extends Line {
    // members of Polygon interface
    // members of Line interface
}
```

```
}
```

Here, the Polygon interface extends the Line interface. Now, if any class implements Polygon, it should provide implementations for all the abstract methods of both Line and Polygon.

====

Advantages of Interface in Java

Now that we know what interfaces are, let's learn about why interfaces are used in Java.

Similar to abstract classes, interfaces help us to achieve abstraction in Java.

Here, we know getArea() calculates the area of polygons but the way area is calculated is different for different polygons. Hence, the implementation of getArea() is independent of one another.

Interfaces provide specifications that a class (which implements it) must follow.

In our previous example, we have used getArea() as a specification inside the interface Polygon. This is like setting a rule that we should be able to get the area of every polygon.

Now any class that implements the Polygon interface must provide an implementation for the getArea() method.

Interfaces are also used to achieve multiple inheritance in Java. For example,

```
interface Line {  
...  
}  
  
interface Polygon {  
...  
}  
  
class Rectangle implements Line, Polygon {  
...  
}
```

Here, the class Rectangle is implementing two different interfaces. This is how we achieve multiple inheritance in Java.

All the methods inside an interface are implicitly `public` and all fields are implicitly `public static final`

All the methods inside an interface are implicitly `public` and all fields are implicitly `public static final`. For example,

```
interface Language {  
    // by default public static final  
    String type = "programming language";  
  
    // by default public  
    void getName();  
}
```

Final

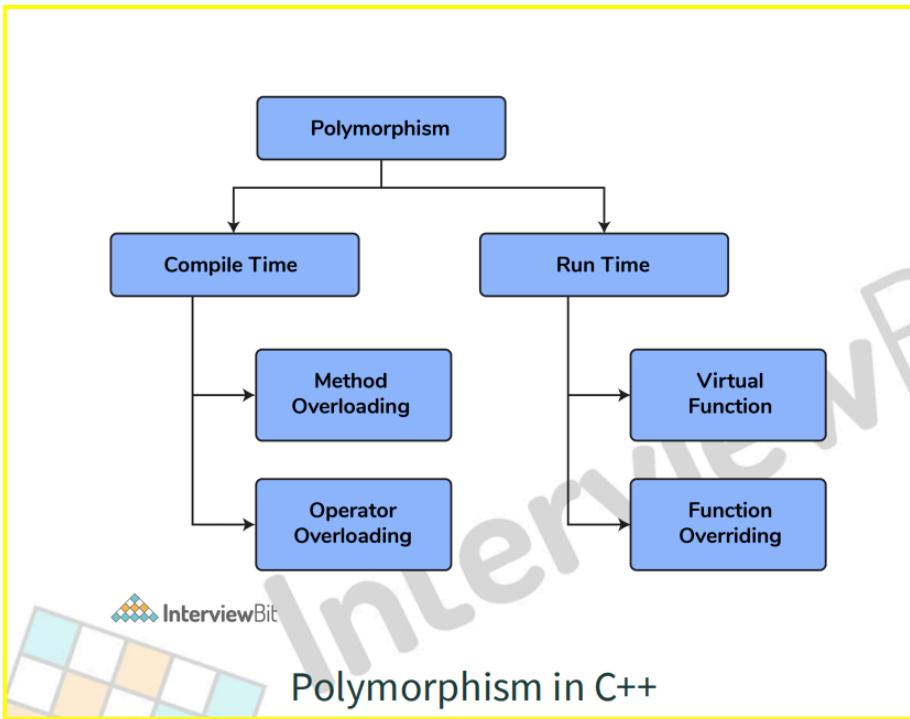
The `final` keyword is a non-access modifier used for classes, attributes and methods, which makes them non-changeable (impossible to inherit or override). The `final` keyword is useful when you want a variable to always store the same value, like PI (3.14159...). The `final` keyword is called a "modifier".

Java Polymorphism

Polymorphism is an important concept of object-oriented programming. It simply means more than one form.

That is, the same entity (method or operator or object) can perform different operations in different scenarios.

Note: The method that is called is determined during the execution of the program. Hence, method overriding is a **run-time polymorphism**.



JDBC

1) What is JDBC?

JDBC is a Java API that is used to connect and execute the query to the database. JDBC API uses JDBC drivers to connect to the database. JDBC API can be used to access tabular data stored into any relational database.

JDBC is an API (Application Programming Interface) that helps a programmer to write a Java program to connect to a database, retrieve the data from the database, and perform various operations on the data in a Java program.

Packages

Java Packages

- ❑ Packages are nothing more than the way we organize files into different directories according to their functionality, usability as well as category they should belong to

- ❑ Files in one directory (or package) would have different functionality from those of another directory.



Example

- ❑ For example: files in `java.io` package do something related to I/O, but files in `java.net` package give us the way to deal with the Network

Name Collision

- ❑ Packaging also help us to avoid class name collision when we use the same class name as that of others
- ❑ The benefits of using package reflect the ease of maintenance, organization, and increase collaboration among developers



The screenshot shows a Windows desktop environment. On the left is a file explorer window displaying a folder structure under 'Computer > New Volume (G:) > Java Programs'. Inside the 'Java Programs' folder, there is a 'world' folder containing a 'HelloWorld.java' file. The file details are: Name: world, Date modified: 17-08-2015 08:05, Type: File folder; Name: HelloWorld.java, Date modified: 17-08-2015 07:57, Type: JAVA File. To the right of the file explorer is a 'Command Prompt' window. The command history is as follows:

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Saurabh>g:
G:\>cd "Java Programs"
G:\Java Programs>javac HelloWorld.java
G:\Java Programs>javac -d . HelloWorld.java
G:\Java Programs>java HelloWorld
Error: Could not find or load main class HelloWorld
G:\Java Programs>java world.HelloWorld
```

How to create package?

- Suppose we have a file called **HelloWorld.java**, and we want to put this file in a package **world**

How to create package?

```
package world;  
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

- Now compile this file as

path> **javac -d . HelloWorld.java**



Remember

We can have only one public class in a single java file.

Name of the file should be same as the name of public class

I

In absence of public class, any class name can be given to the file name.



Static keyword

In Java, a static member refers to a class member (either a field or a method) that belongs to the class itself rather than to instances of the class. This means that the static member is shared among all instances of the class and can be accessed using the class name rather than an instance of the class.

There are two main types of static members in Java:

1. ****Static Variables (Class Variables):**** These are variables declared with the `static` keyword and are shared among all instances of the class. They are associated with the class, not with any specific instance of the class.

```
```java
class Example {
 static int staticVariable = 0;
}
````
```

You can access the static variable using the class name, like `Example.staticVariable` .

2. ****Static Methods:**** These are methods declared with the `static` keyword. They can be called using the class name and are not associated with any particular instance of the class.

```
```java
class Example {
 static void staticMethod() {
 System.out.println("This is a static method.");
 }
}
````
```

You can call the static method using the class name, like `Example.staticMethod()`.

Static members are often used for utility methods, constants, or variables that should be shared among all instances of a class. They are associated with the class itself and not with any specific object created from the class.

Access modifier

Java Access Modifiers

Java supports four categories of accessibility rules

- private
- protected
- public
- default

Modifiers can be used for class, member variables and member functions



With class

- Outer class and inner class
- For **outer class**, there can be only two possibilities, either class is a public class or just a class which means it is of **default type**.
- For **inner class** any among four access modifiers can be used

Remember

- There can be only one public class in a single java file.
- The name of the java file must be the same as the name of the public class.
- Only public class can be accessed directly from outside the package

Member variables and functions

- When members of the class are **private**, they can not be accessed from outside the class body. They are meant to be accessed from the same class in which they are declared.
- When members are **protected**, they can be accessed from any class of the same package and child class from other package
- When members are **public**, they are accessible from any class of any package.
- When members are **default**, they are accessible only from the class of same package.



The `protected` access specifier in Java allows a member (field or method) to be accessed within its own package and by subclasses(Child class), even if they are in different packages. Here's an example:

```
```java
// Animal.java - This class is in the "animals" package
package animals;

public class Animal {
 // Protected field
 protected String species;

 // Protected method
 protected void makeSound() {
 System.out.println("Some generic animal sound");
 }
}
````
```

```
```java
// Dog.java - This class is in the "animals" package
```

```

package animals;

public class Dog extends Animal {
 // Constructor
 public Dog(String species) {
 this.species = species;
 }

 // Overriding the makeSound method
 @Override
 protected void makeSound() {
 System.out.println("Woof! Woof!");
 }

 // Additional method using the protected field
 public void displaySpecies() {
 System.out.println("This is a " + species);
 }
}

```
java
// Main.java - This class can be in a different package
import animals.Dog;

public class Main {
    public static void main(String[] args) {
        // Creating an instance of Dog
        Dog myDog = new Dog("Canine");

        // Accessing the protected field from the subclass
        System.out.println("Species: " + myDog.species);

        // Calling the protected method from the subclass
        myDog.makeSound();

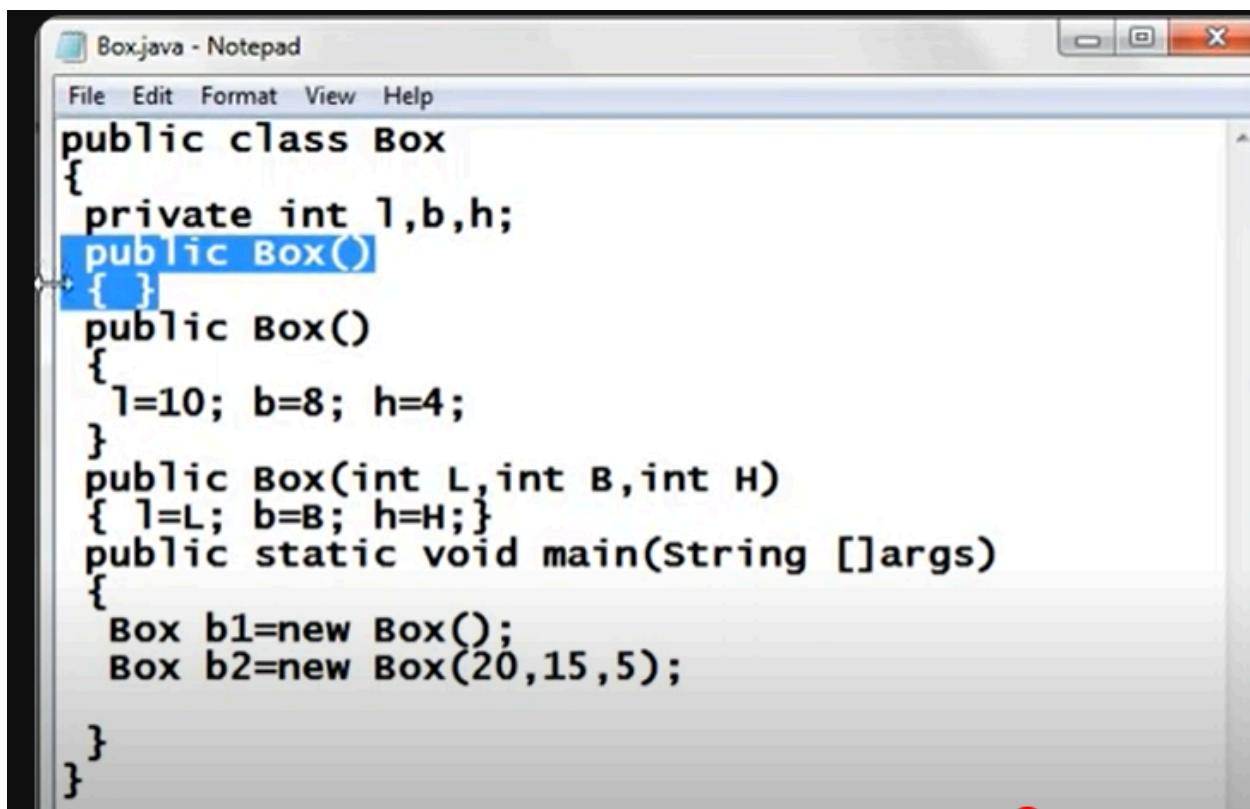
        // Using a method that utilizes the protected field
        myDog.displaySpecies();
    }
}
```

```

In this example, the `Animal` class has a `protected` field `species` and a `protected` method `makeSound`. The `Dog` class, which extends `Animal`, can access and use these protected

members. The `Main` class, which is in a different package, demonstrates how the `Dog` class's protected members can be accessed through an instance of `Dog`.

## constructor



The image shows a screenshot of a Windows-style Notepad window titled "Box.java - Notepad". The window contains the following Java code:

```
public class Box
{
 private int l,b,h;
 public Box()
 {
 }
 public Box()
 {
 l=10; b=8; h=4;
 }
 public Box(int L,int B,int H)
 {
 l=L; b=B; h=H;
 }
 public static void main(String []args)
 {
 Box b1=new Box();
 Box b2=new Box(20,15,5);
 }
}
```

The code defines a class named "Box" with three private integer fields: l, b, and h. It includes two public constructors: one that does nothing and another that initializes l, b, and h to 10, 8, and 4 respectively. A third constructor takes three parameters (L, B, H) and assigns them to l, b, and h. The main method creates two instances of the Box class, b1 and b2, with the second one being initialized with values 20, 15, and 5.

## Constructor is special

---

- A **constructor** is a special method that is used to **initialize a newly created object** and is called implicitly, just after the memory is allocated for the object
- It is **not mandatory** for the coder to write a constructor for the class
- When there is no constructor defined in the class by programmer, compiler implicitly provide a default constructor for the class.



## Constructor Overloading

---

- Constructors can be parameterized
- Constructors can be overloaded

## Inheritance

### Syntax

```
class SubClass extends SuperClass
{
}
```

- extends** is a keyword
- Base class means Super Class
- Derived Class means Sub Class

## Remember

---

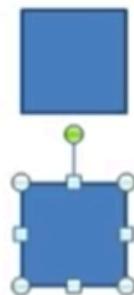
- ❑ In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of *subclasses*
- ❑ Private members of the superclass are not accessible by the subclass and can only be indirectly accessed.
- ❑ Members that have default accessibility in the superclass are also not accessible by subclasses in other packages



## Java Supports

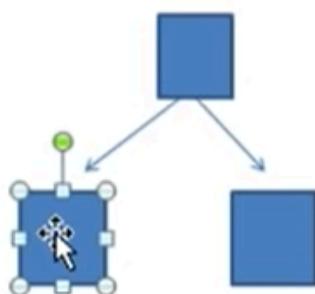
---

- ❑ Single Inheritance
- ❑ Multilevel Inheritance
- ❑ Hierarchical Inheritance



# Java Supports

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance



```
// Parent class (superclass)
class Animal {
 String name;

 Animal(String name) {
 this.name = name;
 }

 void eat() {
 System.out.println(name + " is eating.");
 }

 void sleep() {
 System.out.println(name + " is sleeping.");
 }
}

// Child class (subclass) inheriting from Animal
class Dog extends Animal {
 Dog(String name) {
 super(name); // calling the constructor of the superclass
 }
}
```

```
// Additional method specific to Dog
void bark() {
 System.out.println(name + " is barking.");
}
}

public class InheritanceExample {
 public static void main(String[] args) {
 // Creating an instance of the subclass (Dog)
 Dog myDog = new Dog("Buddy");

 // Accessing methods from the superclass (Animal)
 myDog.eat();
 myDog.sleep();

 // Accessing method from the subclass (Dog)
 myDog.bark();
 }
}
```

## Instance Initialization Block

- ❑ An *instance initializer or Initialization block* declared in a class is executed when an instance of the class is created
- ❑ `return` keyword cannot be used in Initialization block
- ❑ Instance initializers are permitted to refer to the current object via the keyword `this` and to use the keyword `super`



## Static initialization block

```
public class Test
{
 private static int k;
 static
 {
 System.out.println("Static Initialization Block: k="+k);
 k=10;
 }
 public static void main(String []args)
 {
 new Test();
 }
}
```



Saquib Shukla Sir

## Static initialization block

- ❑ A *static initializer* declared in a class is executed when the class is initialized
- ❑ Static initializers may be used to initialize the class variables of the class
- ❑ `return` keyword cannot be used in static Initialization block
- ❑ `this` or `super` can not be used in static block

## super Keyword in Java Inheritance

Previously we saw that the same method in the subclass overrides the method in superclass.

In such a situation, the `super` keyword is used to call the method of the parent class from the method of the child class.

```
class Animal {

 // method in the superclass
 public void eat() {
 System.out.println("I can eat");
 }
}

// Dog inherits Animal
class Dog extends Animal {

 // overriding the eat() method
 @Override
 public void eat() {

 // call method of superclass
 super.eat(); // vvi
 System.out.println("I eat dog food");
 }

 // new method in subclass
 public void bark() {
 System.out.println("I can bark");
 }
}

class Main {
 public static void main(String[] args) {

 // create an object of the subclass
 Dog labrador = new Dog();

 // call the eat() method
 labrador.eat();
 }
}
```

```
 labrador.bark();
}
}
```

## Output

```
I can eat
I eat dog food
I can bark
```

In the above example, the `eat()` method is present in both the base class `Animal` and the derived class `Dog`. Notice the statement,

```
super.eat();
```

Here, the `super` keyword is used to call the `eat()` method present in the superclass.

We can also use the `super` keyword to call the constructor of the superclass from the constructor of the subclass.

- **Method overriding is also known as runtime polymorphism. Hence, we can achieve Polymorphism( more than 1 form) in Java with the help of inheritance.**

The method that is called is determined during the execution of the program. Hence, method overriding is a run-time polymorphism.

## Function Overloading in Java

- ❑ If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but signatures that are not same, then the method name is said to be *overloaded*.
- ❑ Method overloading is a way to implement polymorphism



The screenshot shows a Java code editor with the file 'Example1.java' open. The code defines a class hierarchy where class B extends class A and implements two methods named f1 with different signatures. The code also includes a main method to demonstrate object creation and method invocation.

```
1 class A
2 {
3 }
4 class B extends A
5 {
6 }
7 public void f1(int x)
8 {
9 System.out.println("Class A");
10 }
11 public void f1(int x,int y)
12 {
13 System.out.println("Class B");
14 }
15
16 public class Example1
17 {
18 public static void main(String[]args)
19 {
20 B obj=new B();
21 obj.f1(1);
22 obj.f1(1,2);
23 }
}
```

The screenshot shows a Java code editor window titled "Example1.java". The code defines two classes, A and B, and a main class Example1.

```
1 class A //Car
2 {
3 public void f1(int x)
4 {
5 System.out.println("Class A");
6 }
7 }
8 class B extends A //SportsCar
9 {
10
11 public void f1(int x)
12 {
13 System.out.println("Class B");
14 }
15 }
16 public class Example1
17 {
18 public static void main(String[]args)
19 {
20 B obj=new B();
21 obj.f1(5);
22 }
23 }
```

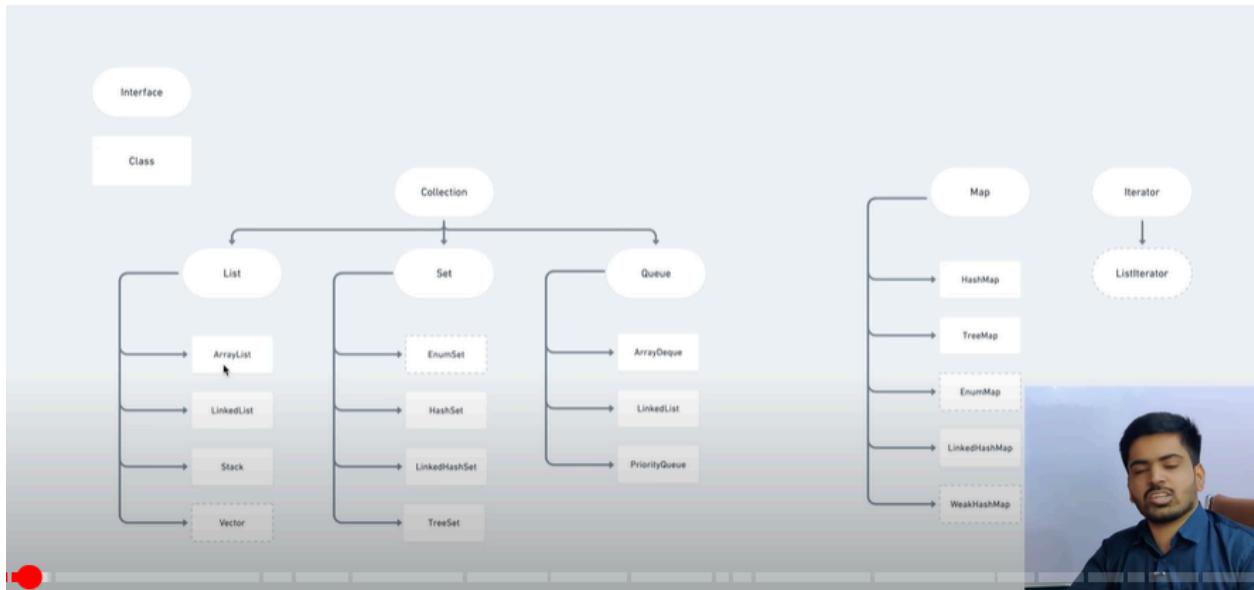
The code uses the `System.out.println` statement to print the class name when the `f1` method is called. In the `main` method, an object of class `B` is created and its `f1` method is called with the argument `5`. The output will be "Class B".

Overloading -overriding

# Collections

## ArrayList

list



The **Collection in Java** (like stl in cpp) is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many (main thing)**interfaces (Set, List, Queue, Deque)** and classes (**ArrayList**, **Vector**, **LinkedList**, **PriorityQueue**, **HashSet**, **LinkedHashSet**, **TreeSet**).

A screenshot of the IntelliJ IDEA IDE interface. The left pane shows a project structure for 'LearnCollectionFramework' with a 'src' folder containing 'LearnArrayList.java'. The right pane displays the Java code for 'LearnArrayList'. The code creates an ArrayList of integers, adds elements 1, 2, 3, and 4, and then prints the list. The output window at the top right shows the printed list: [1, 2, 3] and [1, 2, 3, 4]. Below the code editor, there is a video frame showing a man with a beard and dark hair, wearing a blue shirt, speaking.

```
import java.util.ArrayList;
import java.util.List;

public class LearnArrayList {

 public static void main(String[] args) {

 List<Integer> list = new ArrayList();
 list.add(1);
 list.add(2);
 list.add(3);
 System.out.println(list);

 list.add(4); // This will add 4 at the end of the List
 System.out.println(list);

 }
}
```

## Functionalities

A screenshot of the IntelliJ IDEA IDE interface, similar to the previous one but with a different code snippet. The code demonstrates the use of the `list.add(index, element)` method. It creates an ArrayList, adds elements 1, 2, 3, and then adds element 50 at index 1. The output window shows the list: [1, 2, 3] and [1, 50, 2, 3, 4]. Below the code editor, there is a video frame showing a man with a beard and dark hair, wearing a blue shirt, speaking.

```
import java.util.ArrayList;
import java.util.List;

public class LearnArrayList {

 public static void main(String[] args) {

 List<Integer> list = new ArrayList();
 list.add(1);
 list.add(2);
 list.add(3);
 System.out.println(list);

 list.add(4); // This will add 4 at the end of the List
 System.out.println(list);

 list.add(1, 50); // This will add 50 at index 1
 System.out.println(list);

 }
}
```

A screenshot of the IntelliJ IDEA IDE. The left pane shows a project structure for 'LearnCollectionFramework' with a 'src' folder containing 'LearnArrayList.java'. The right pane shows the code for 'LearnArrayList.java' and the 'Run' tool window.

```
public static void main(String[] args) {
 List<Integer> list = new ArrayList();
 list.add(1);
 list.add(2);
 list.add(3);
 System.out.println(list);

 list.add(4); // This will add 4 at the end of the List
 System.out.println(list);

 list.add(index: 1, element: 50); // This will add 50 at index 1
 System.out.println(list);

 List<Integer> newList = new ArrayList(); // Creating a new list
 newList.add(150);
 newList.add(160);

 list.addAll(newList); // This will add all the elements of newList to list
 System.out.println(list);

 System.out.println(list.get(1));
}
```

The 'Run' tool window shows the output of the program:

```
[1, 2, 3]
[1, 2, 3, 4]
[1, 50, 2, 3, 4]
[1, 50, 2, 3, 4, 150, 160]
50
```

Process finished with exit code 0

## types of iteration in collection

A screenshot of the IntelliJ IDEA IDE. The left pane shows a project structure for 'LearnCollectionFramework' with a 'src' folder containing 'LearnArrayList.java'. The right pane shows the code for 'LearnArrayList.java' and the 'Run' tool window.

```
list.add(30);
list.add(40);
list.add(50);
list.add(60);
list.add(70);
list.add(80);

System.out.println(list);

for (int i = 0; i < list.size(); i++) {
 System.out.println("the element is " + list.get(i));
}

for (Integer element: list) {
 System.out.println("foreach element is " + element);
}

Iterator<Integer> it = list.iterator();

while (it.hasNext()) {
 System.out.println("iterator " + it.next());
}

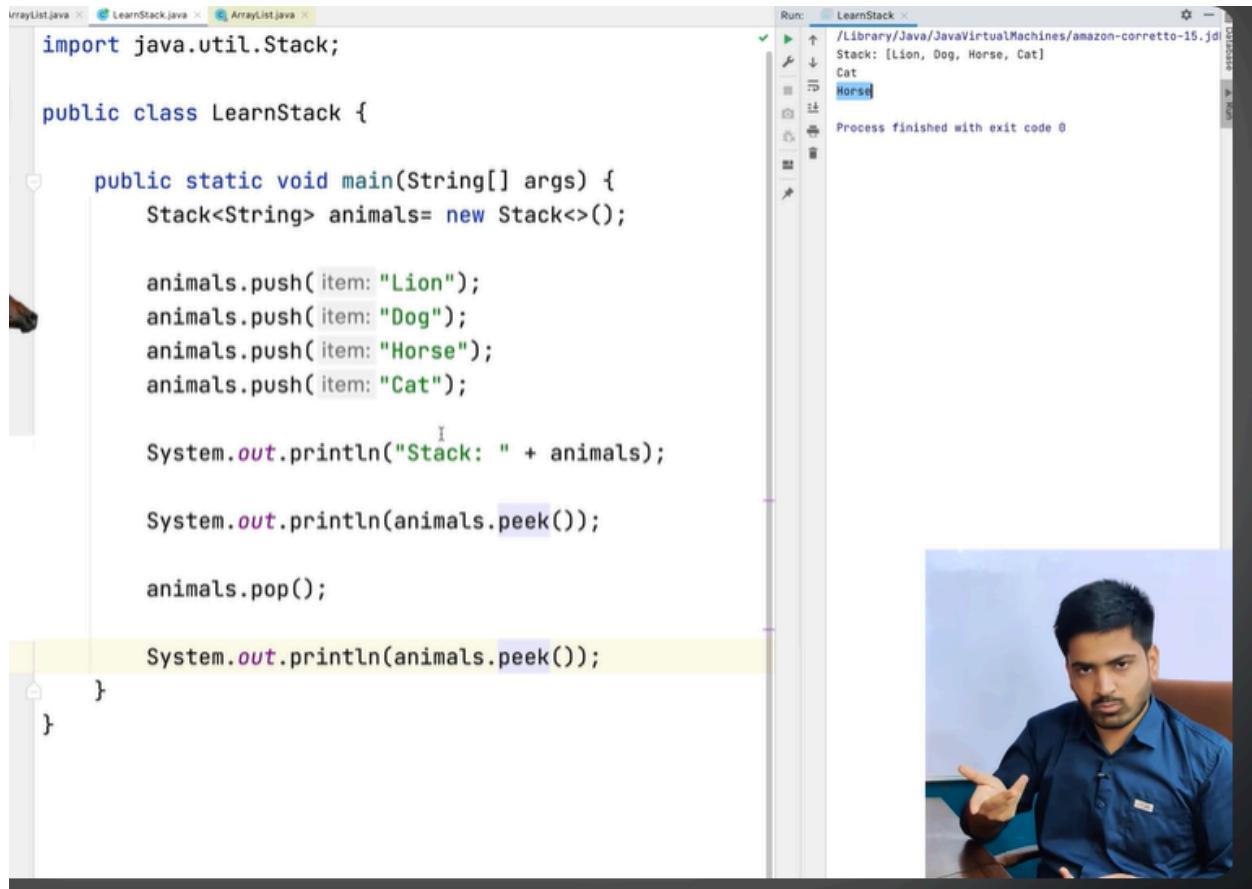
list.set(2, 1000);
```

The 'Run' tool window shows the output of the program:

```
the element is 10
the element is 20
the element is 30
the element is 40
the element is 50
the element is 60
the element is 70
the element is 80
foreach element is 10
foreach element is 20
foreach element is 30
foreach element is 40
foreach element is 50
foreach element is 60
foreach element is 70
foreach element is 80
iterator 10
iterator 20
iterator 30
iterator 40
iterator 50
iterator 60
iterator 70
iterator 80
```

Process finished with exit code 0

## stacks



A screenshot of a Java IDE interface. On the left, there are three tabs: 'ArrayList.java', 'LearnStack.java', and 'ArrayList.java'. The 'LearnStack.java' tab is active, displaying the following code:

```
import java.util.Stack;

public class LearnStack {

 public static void main(String[] args) {
 Stack<String> animals= new Stack<>();

 animals.push(item: "Lion");
 animals.push(item: "Dog");
 animals.push(item: "Horse");
 animals.push(item: "Cat");

 System.out.println("Stack: " + animals);

 System.out.println(animals.peek());

 animals.pop();

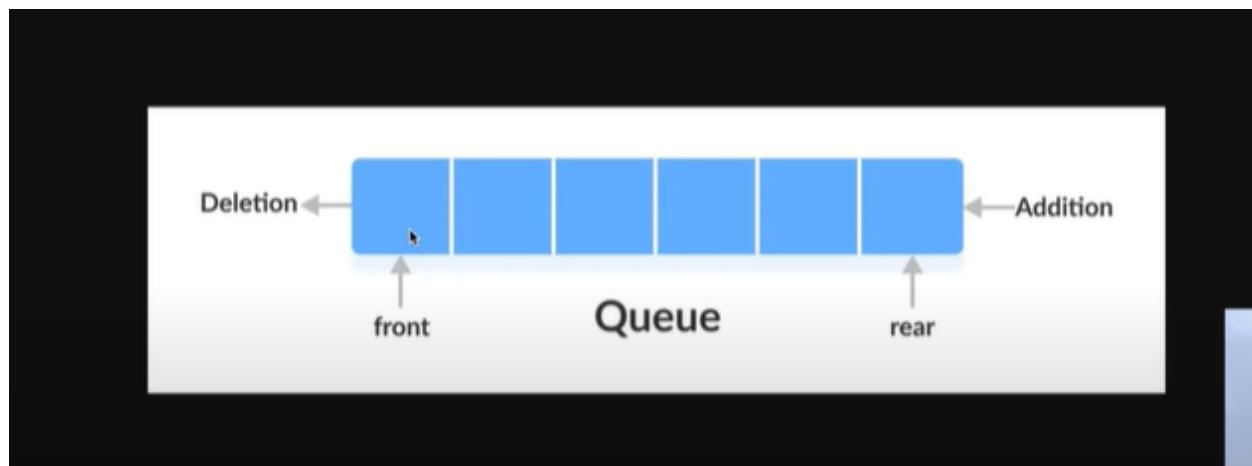
 System.out.println(animals.peek());
 }
}
```

The 'Run' tab on the right shows the output of the program:

```
Run: LearnStack
Stack: [Lion, Dog, Horse, Cat]
Cat
Horse
Process finished with exit code 0
```

On the right side of the IDE, there is a video player window showing a man in a blue shirt speaking.

## Queue



Some of the commonly used methods of the `Queue` interface are:

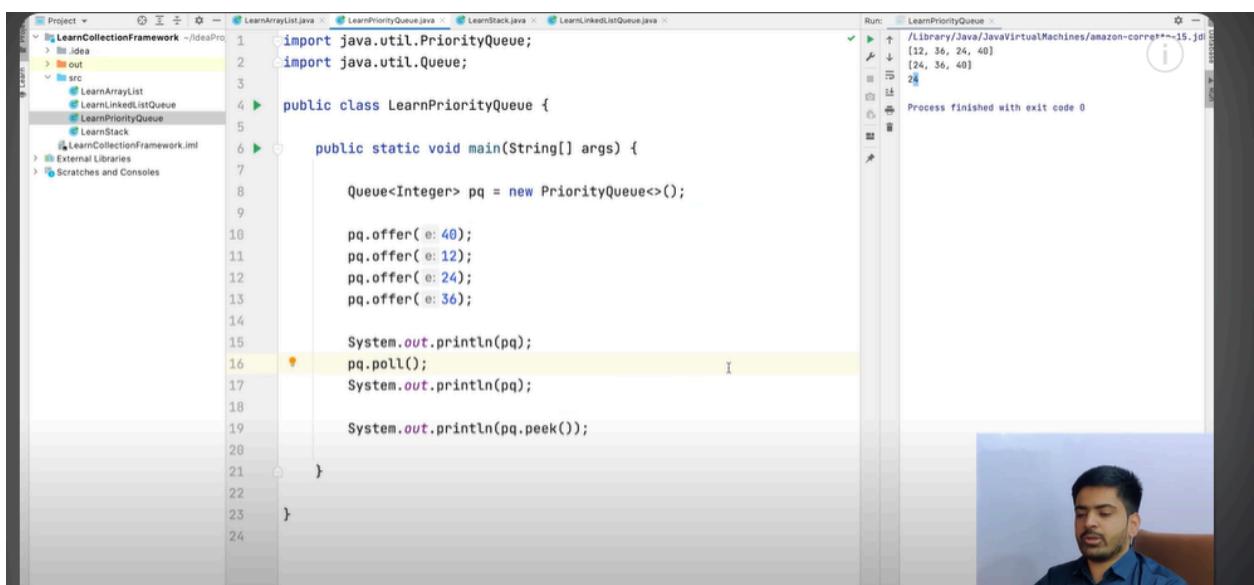
- `add()` - Inserts the specified element into the queue. If the task is successful, `add()` returns `true`, if not it throws an exception.
- `offer()` - Inserts the specified element into the queue. If the task is successful, `offer()` returns `true`, if not it returns `false`.
- `element()` - Returns the head of the queue. Throws an exception if the queue is empty.
- `peek()` - Returns the head of the queue. Returns `null` if the queue is empty.
- `remove()` - Returns and removes the head of the queue. Throws an exception if the queue is empty.
- `poll()` - Returns and removes the head of the queue. Returns `null` if the queue is empty.

## Implementation of the Queue Interface

### 1. Implementing the LinkedList Class



Priority Queue—implements min heap



```

import java.util.PriorityQueue;
import java.util.Queue;

public class LearnPriorityQueue {
 public static void main(String[] args) {
 Queue<Integer> pq = new PriorityQueue<>();
 pq.offer(e: 40);
 pq.offer(e: 12);
 pq.offer(e: 24);
 pq.offer(e: 36);

 System.out.println(pq);
 pq.poll();
 System.out.println(pq);

 System.out.println(pq.peek());
 }
}

```

Max heap



A screenshot of an IDE (IntelliJ IDEA) showing Java code for a priority queue. The code imports java.util.Comparator, java.util.PriorityQueue, and java.util.Queue, and defines a main method that creates a PriorityQueue with reverse order, offers elements 40, 36, 24, and 12, polls elements, and prints the peeked element. The run output shows the elements in descending order: [40, 36, 24, 12], [36, 12, 24], and 36.

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;

public class LearnPriorityQueue {

 public static void main(String[] args) {

 Queue<Integer> pq = new PriorityQueue<>(<comparator>.reverseOrder());

 pq.offer(e: 40);
 pq.offer(e: 12);
 pq.offer(e: 24);
 pq.offer(e: 36);

 System.out.println(pq);
 pq.poll();
 System.out.println(pq);

 System.out.println(pq.peek());
 }
}
```



A screenshot of an IDE (IntelliJ IDEA) showing Java code for a priority queue. The code imports java.util.Comparator, java.util.PriorityQueue, and java.util.Queue, and defines a main method that creates a PriorityQueue with reverse order, offers elements 40, 36, 24, and 12, polls elements, and prints the peeked element. The run output shows the elements in descending order: [40, 36, 24, 12], [36, 12, 24], and 36.

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;

public class LearnPriorityQueue {

 public static void main(String[] args) {

 Queue<Integer> pq = new PriorityQueue<>(<comparator>.reverseOrder());

 pq.offer(e: 40);
 pq.offer(e: 12);
 pq.offer(e: 24);
 pq.offer(e: 36);

 System.out.println(pq);
 pq.poll();
 System.out.println(pq);

 System.out.println(pq.peek());
 }
}
```

Array deque

```
5 ► 5 public static void main(String[] args) { /Library/Java/JavaVirtualMachines/amazon-corretto
6
7 ArrayDeque<Integer> adq = new ArrayDeque<>(); [12, 23, 45, 26]
8
9 adq.offer(e: 23); 12
10 adq.offerFirst(e: 12); 26
11 adq.offerLast(e: 45); 12
12 adq.offer(e: 26); poll() [23, 45, 26]
13
14 System.out.println(adq); 23
15
16 System.out.println(adq.peek()); pollFirst() [45, 26]
17 System.out.println(adq.peekFirst()); 26
18 System.out.println(adq.peekLast()); pollLast() [45]
19
20 System.out.println(adq.poll()); Process finished with exit code 0
21 System.out.println("poll() " +adq);
22
23 System.out.println(adq.pollFirst()); 12
24 System.out.println("pollFirst() " +adq);
25
26 System.out.println(adq.pollLast()); 26
27 System.out.println("pollLast() " +adq);
28
29 }
30
31 }
```

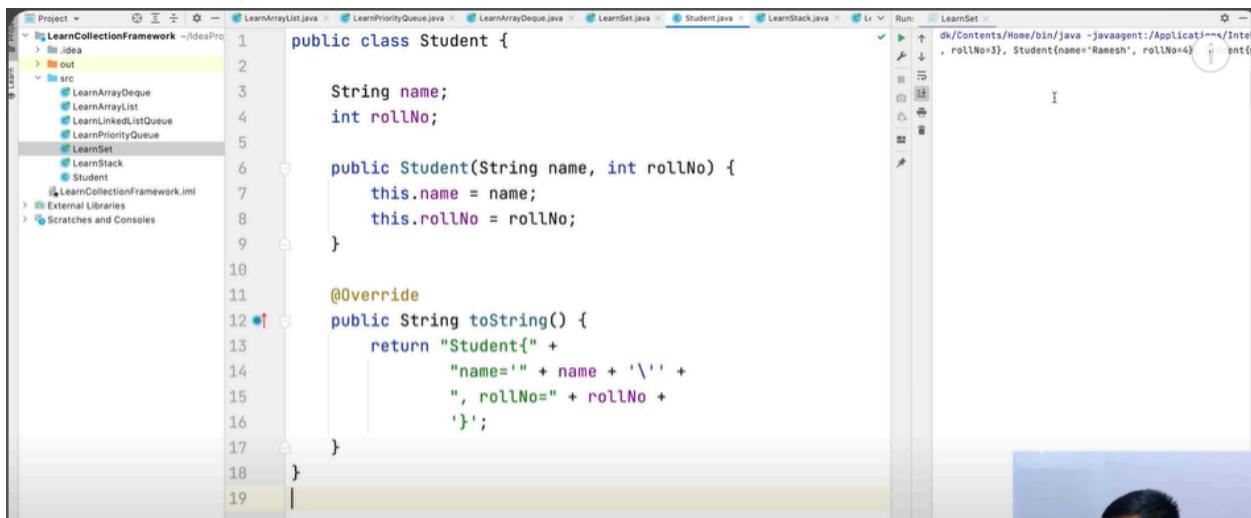


## Hashset

```
nCollectionFramework ~/ideaPro 6 ► 6 public static void main(String[] args) { /Library/Java/JavaVirtualMachines/amazon-corretto-15.jdi
lea
it
c
LearnArrayDeque
LearnArrayList
LearnLinkedListQueue
LearnPriorityQueue
LearnSet
LearnStack
arnCollectionFramework.iml
nal Libraries
tches and Consoles
10 Set<Integer> set = new HashSet<>();
11
12 set.add(32);
13 set.add(2);
14 set.add(54);
15 set.add(21);
16 set.add(65);
17
18 System.out.println(set);
19
20 set.remove(o: 54);
21
22 System.out.println(set);
23
24 System.out.println(set.contains(21));
25
26 System.out.println(set.isEmpty());
27
28 System.out.println(set.size());
29
30 }
31
32 }
```



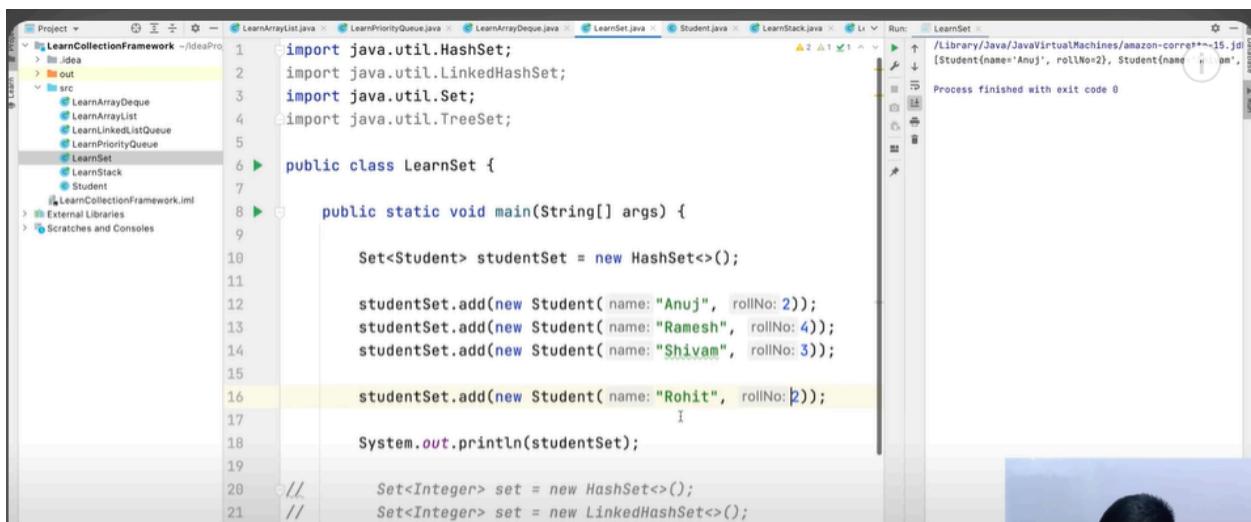
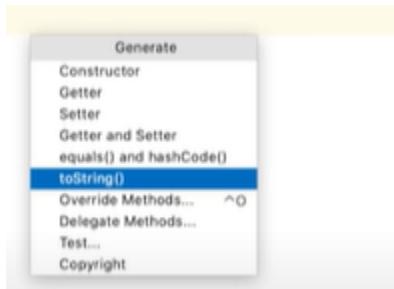
## Treeset->harshset



```
public class Student {
 String name;
 int rollNo;

 public Student(String name, int rollNo) {
 this.name = name;
 this.rollNo = rollNo;
 }

 @Override
 public String toString() {
 return "Student{" +
 "name='" + name + '\'' +
 ", rollNo=" + rollNo +
 '}';
 }
}
```



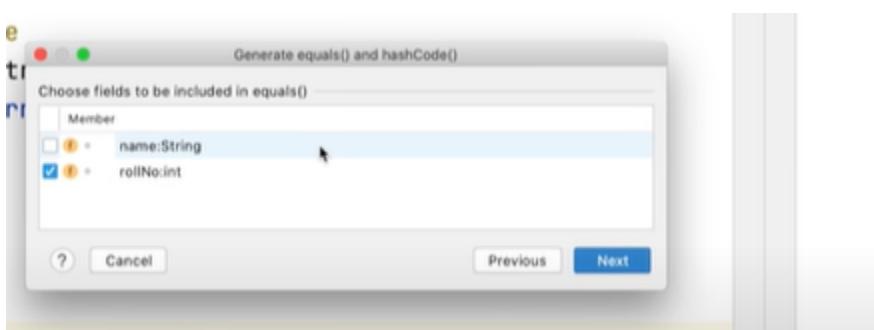
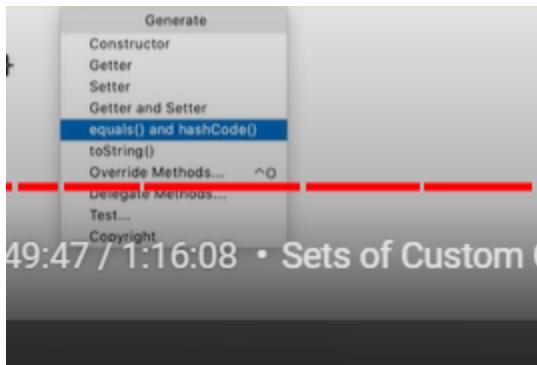
```
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;
import java.util.TreeSet;

public class LearnSet {

 public static void main(String[] args) {
 Set<Student> studentSet = new HashSet<>();

 studentSet.add(new Student(name: "Anuj", rollNo: 2));
 studentSet.add(new Student(name: "Ramesh", rollNo: 4));
 studentSet.add(new Student(name: "Shivam", rollNo: 3));

 studentSet.add(new Student(name: "Rohit", rollNo: 2));
 System.out.println(studentSet);
 }
}
```



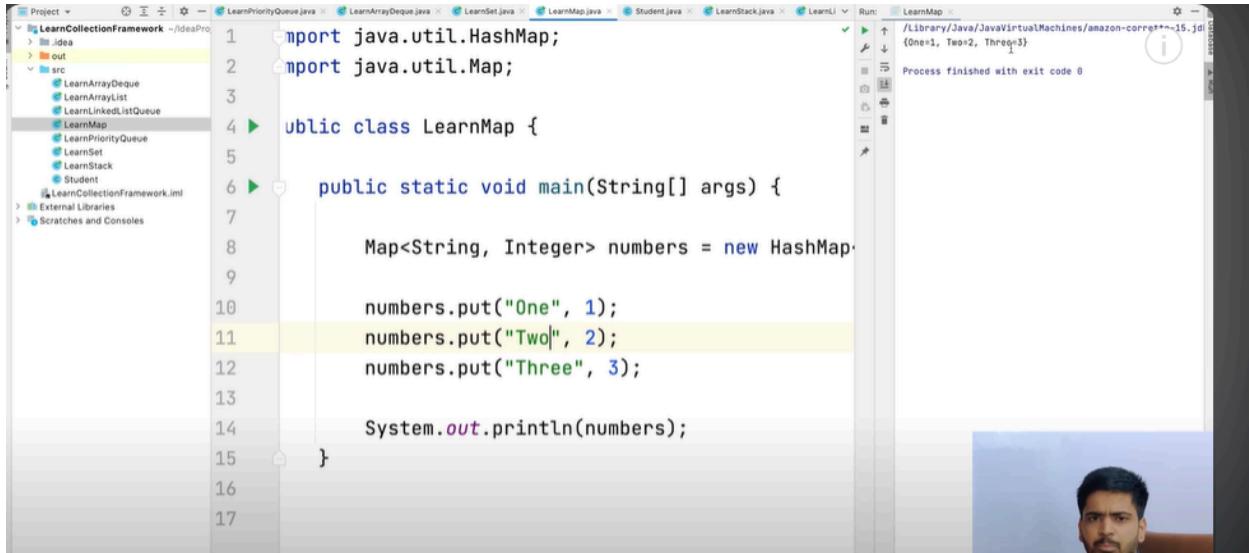
## Hashcode and equals

The screenshot shows the Java IDE with the 'Student.java' file open. The code defines a 'Student' class with three methods: `toString()`, `equals()`, and `hashCode()`. The `toString()` method returns a string representation of the student with fields 'name' and 'rollNo'. The `equals()` method checks if the object is null or has the same class, then compares the 'rollNo' field. The `hashCode()` method returns the hash code of the 'rollNo' field. A video overlay of a man speaking is visible on the right side of the screen.

```
12
13
14 @Override
15 public String toString() {
16 return "Student{" +
17 "name='" + name + '\'' +
18 ", rollNo=" + rollNo +
19 '}';
20
21 @Override
22 public boolean equals(Object o) {
23 if (this == o) return true;
24 if (o == null || getClass() != o.getClass()) return false;
25 Student student = (Student) o;
26 return rollNo == student.rollNo;
27 }
28
29 @Override
30 public int hashCode() {
31 return Objects.hash(rollNo);
32 }
33
34 }
```

# Map

Store value in key value pair



A screenshot of a Java IDE (IntelliJ IDEA) showing a code editor with a Java file named `LearnMap.java`. The code creates a `HashMap` and adds three entries: "One" with value 1, "Two" with value 2, and "Three" with value 3. The output window shows the printed map with these entries.

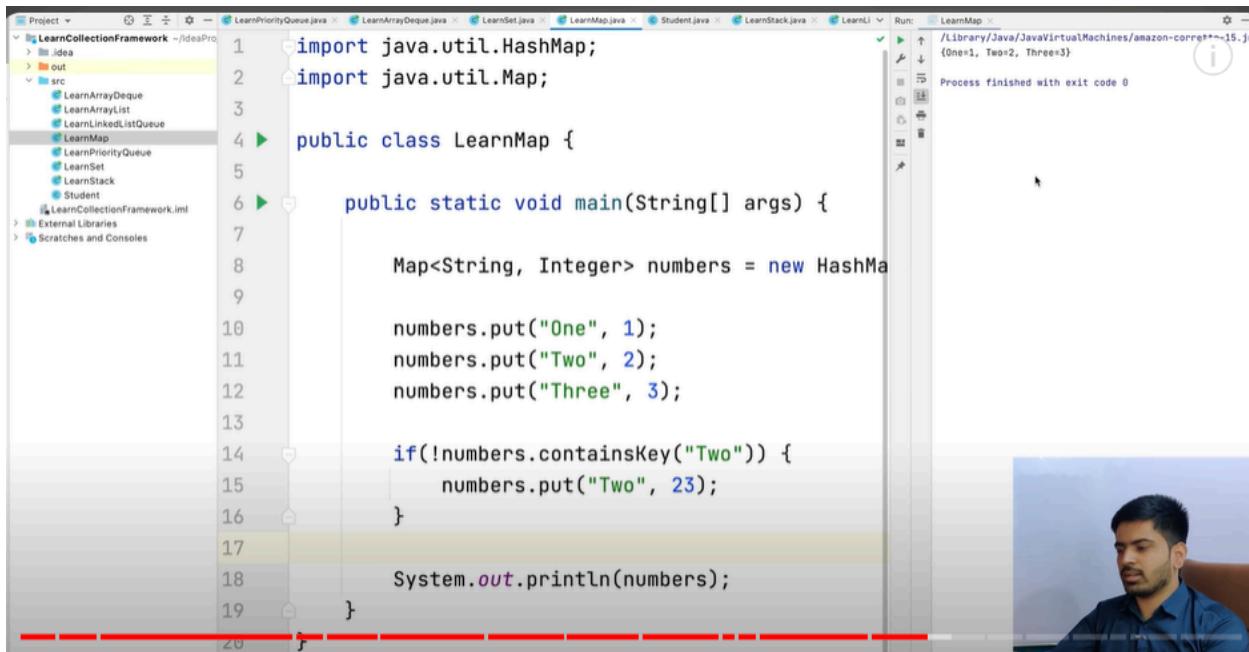
```
import java.util.HashMap;
import java.util.Map;

public class LearnMap {

 public static void main(String[] args) {
 Map<String, Integer> numbers = new HashMap<>();

 numbers.put("One", 1);
 numbers.put("Two", 2);
 numbers.put("Three", 3);

 System.out.println(numbers);
 }
}
```



A screenshot of a Java IDE (IntelliJ IDEA) showing a code editor with a Java file named `LearnMap.java`. The code creates a `HashMap`, adds entries for "One", "Two", and "Three", and then checks if "Two" is a key. If it is not, it adds a new entry "Two" with value 23. The output window shows the printed map with all four entries: "One", "Two", "Three", and "Two" with value 23.

```
import java.util.HashMap;
import java.util.Map;

public class LearnMap {

 public static void main(String[] args) {
 Map<String, Integer> numbers = new HashMap<>();

 numbers.put("One", 1);
 numbers.put("Two", 2);
 numbers.put("Three", 3);

 if(!numbers.containsKey("Two")) {
 numbers.put("Two", 23);
 }

 System.out.println(numbers);
 }
}
```

```
numbers.putIfAbsent("Two", 23);
System.out.println(numbers);

for (Map.Entry<String, Integer> e: numbers.entrySet()) {
 System.out.println(e);
}

System.out.println(e.getKey());
System.out.println(e.getValue());
}

for (String key: numbers.keySet()) {
 System.out.println(key);
}

for(Integer value: numbers.values()) {
 System.out.println(value);
}
```

```
{four=4, One=1, Two=2, five=5, Three=3}
four
4
One=1
One
1
Two=2
Two
2
five=5
five
5
Three=3
Three
3
```

Process finished with exit code 0



```
numbers.putIfAbsent("Two", 23);
System.out.println(numbers);

for (Map.Entry<String, Integer> e: numbers.entrySet()) {
 System.out.println(e);

 System.out.println(e.getKey());
 System.out.println(e.getValue());
}

for (String key: numbers.keySet()) {
 System.out.println(key);
}

for(Integer value: numbers.values()) {
 System.out.println(value);
}
```

```
{four=4, One=1, Two=2, five=5, Three=3}
four
4
One=1
One
1
Two=2
Two
2
five=5
five
5
Three=3
Three
3
```

Process finished with exit code 0



```
numbers.put("Three", 3);
numbers.put("four", 4);
numbers.put("five", 5);

// if(!numbers.containsKey("Two")) {
// numbers.put("Two", 23);
// }

// numbers.putIfAbsent("Two", 23);
System.out.println(numbers);
System.out.println(numbers.containsValue(23));

System.out.println(numbers.isEmpty());

// for (Map.Entry<String, Integer> e: numbers.entrySet()) {
// System.out.println(e);
// }

// System.out.println(e.getKey());
// System.out.println(e.getValue());
```



The code editor shows Java code with syntax highlighting. A yellow highlight covers the line `System.out.println(numbers.isEmpty());`. A red underline is placed under the entire block of code starting with `//` and ending with the final `});` line. The output window above the code shows the results of the execution.

```
/Library/Java/JavaVirtualMachines/amazon-corretto-11.0.11.jdk/Contents/Home/bin/java -jar /Library/Java/JavaVirtualMachines/amazon-corretto-11.0.11.jdk/Contents/Home/lib/jvm/jrt-fs.jar --list=java.util.Map
{four=4, One=1, Two=2, five=5, Three=3}
true
false

Process finished with exit code 0
```

```
ic class LearnMap {

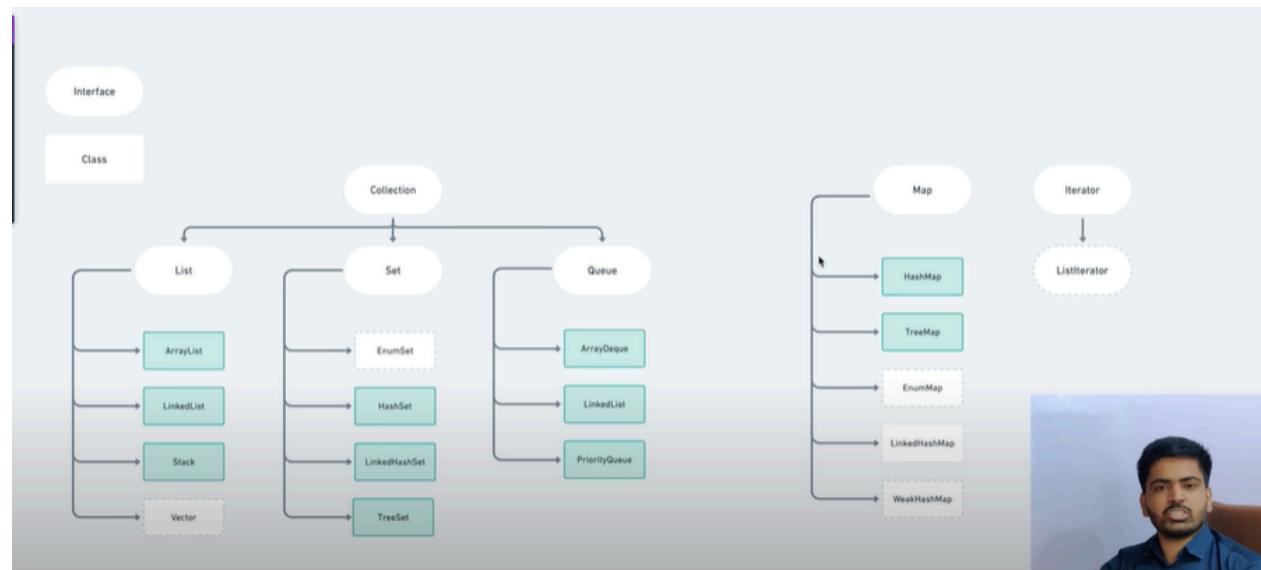
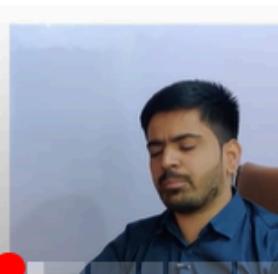
public static void main(String[] args) {

 Map<String, Integer> numbers = new TreeMap<>()

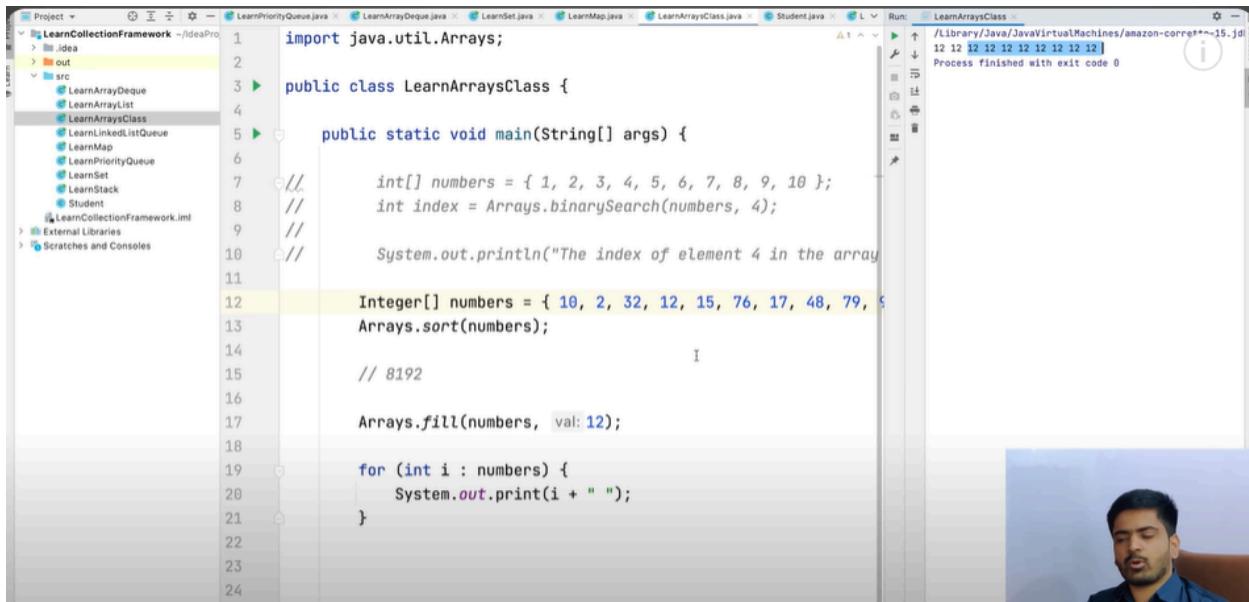
 numbers.put("One", 1);
 numbers.put("Two", 2);
 numbers.put("Three", 3);
 numbers.put("Four", 4);
 numbers.put("Five", 5);

 numbers.remove(key: "Three");

 if(!numbers.containsKey("Two")) {
 numbers.put("Two", 23);
 }
}
```

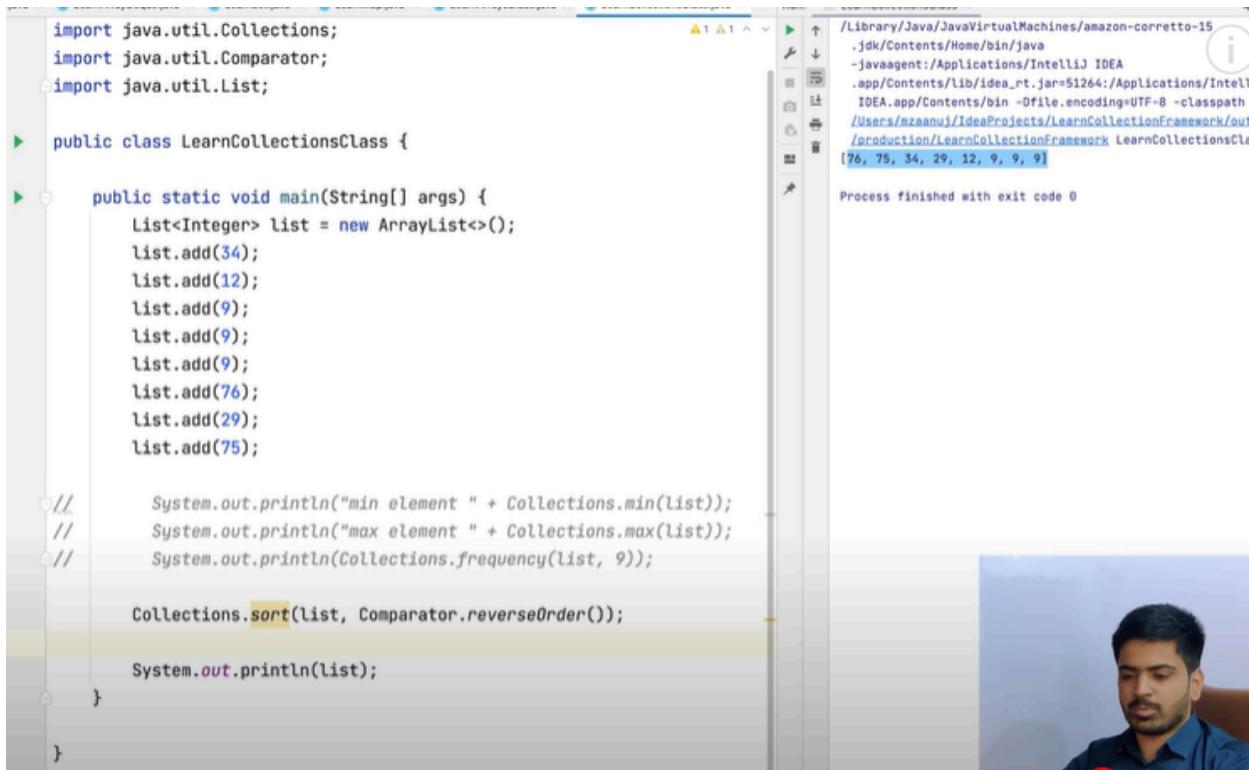


## ArrayClass



Screenshot of IntelliJ IDEA showing the code for `LearnArraysClass.java`. The code uses `java.util.Arrays` to demonstrate binary search and sorting. A video player window is visible on the right side of the interface.

```
import java.util.Arrays;
public class LearnArraysClass {
 public static void main(String[] args) {
 int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
 int index = Arrays.binarySearch(numbers, 4);
 System.out.println("The index of element 4 in the array");
 Integer[] numbers = { 10, 2, 32, 12, 15, 76, 17, 48, 79, 8 };
 Arrays.sort(numbers);
 // 8192
 Arrays.fill(numbers, val: 12);
 for (int i : numbers) {
 System.out.print(i + " ");
 }
 }
}
```



Screenshot of IntelliJ IDEA showing the code for `LearnCollectionsClass.java`. The code uses `java.util.Collections` to demonstrate list operations like min, max, and frequency. A video player window is visible on the right side of the interface.

```
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class LearnCollectionsClass {

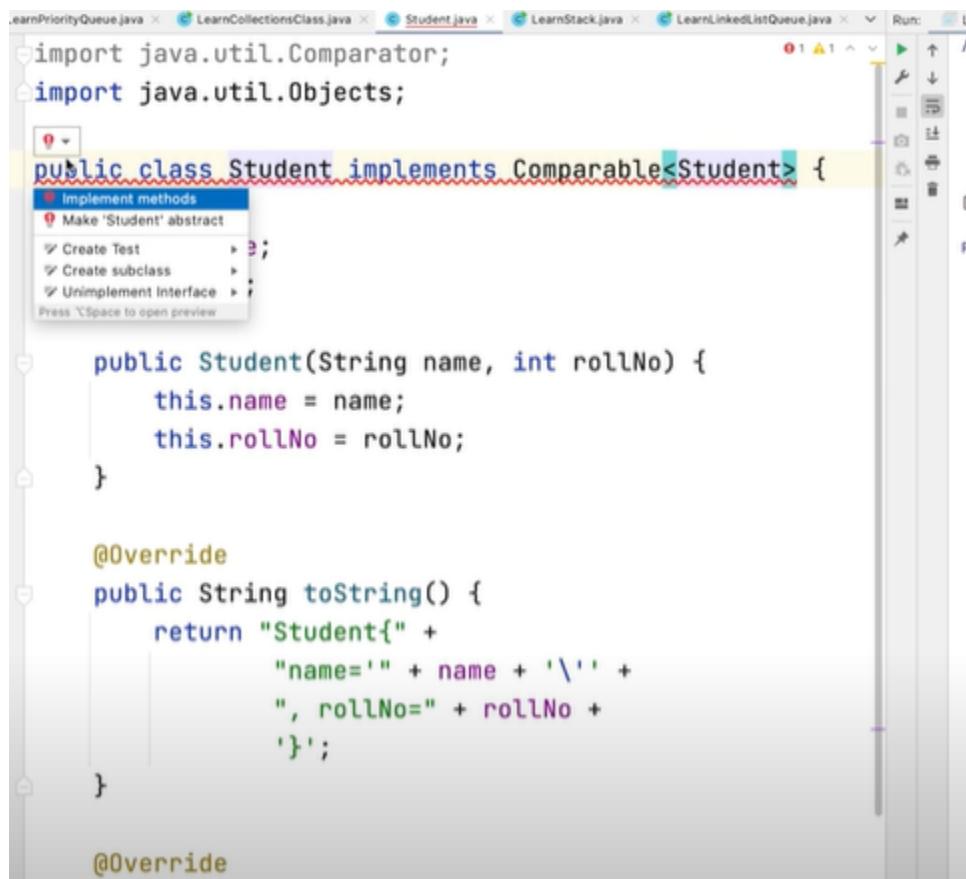
 public static void main(String[] args) {
 List<Integer> list = new ArrayList<>();
 list.add(34);
 list.add(12);
 list.add(9);
 list.add(9);
 list.add(9);
 list.add(76);
 list.add(29);
 list.add(75);

 System.out.println("min element " + Collections.min(list));
 System.out.println("max element " + Collections.max(list));
 System.out.println(Collections.frequency(list, 9));

 Collections.sort(list, Comparator.reverseOrder());

 System.out.println(list);
 }
}
```

*imp*



The screenshot shows the IntelliJ IDEA interface with several tabs at the top: LearnPriorityQueue.java, LearnCollectionsClass.java, Student.java, LearnStack.java, and LearnLinkedListQueue.java. The Student.java tab is active. A tooltip from the IDE's code completion feature is displayed over the line "public class Student implements Comparable<Student> {". The tooltip contains the following options:

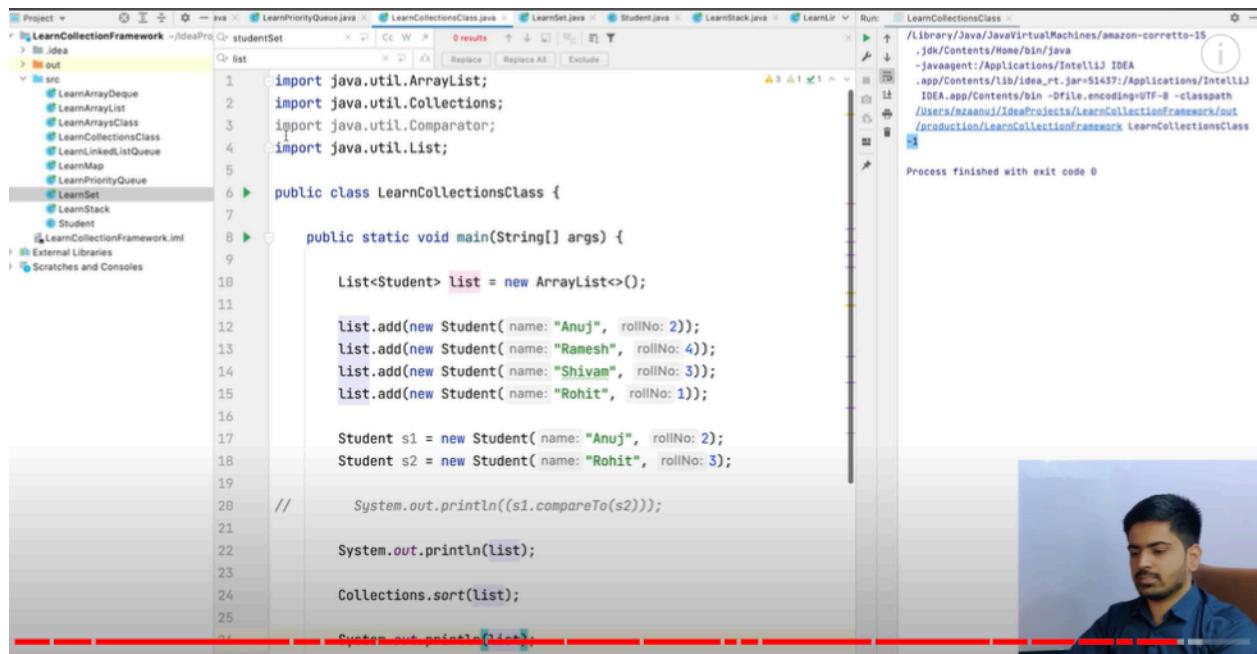
- Implement methods
- Create Test
- Create subclass
- Unimplement Interface

The main code area shows the implementation of the Comparable interface:

```
import java.util.Comparator;
import java.util.Objects;

public class Student implements Comparable<Student> {
 public Student(String name, int rollNo) {
 this.name = name;
 this.rollNo = rollNo;
 }

 @Override
 public String toString() {
 return "Student{" +
 "name='" + name + '\'' +
 ", rollNo=" + rollNo +
 '}';
 }
}
```



The screenshot shows the IntelliJ IDEA interface with the Project tool window on the left and the Run tool window on the right. The Project tool window shows a file structure for a project named "LearnCollectionFramework". The Run tool window shows the command line for running the "LearnCollectionsClass" application:

```
/Library/Java/JavaVirtualMachines/amazon-corretto-15.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA.app/Contents/lib/idea_rt.jar=51457:/Applications/IntelliJ IDEA.app/Contents/bin -Dfile.encoding=UTF-8 -classpath /Users/mzaanuj/IdeaProjects/learnCollectionFramework/out/production/LearnCollectionFramework LearnCollectionsClass
```

The status bar at the bottom right indicates "Process finished with exit code 0". In the bottom right corner of the interface, there is a small video camera icon showing a person's face.

Comparator-more than 2 class

The screenshot shows the IntelliJ IDEA interface with the code editor open. The code is as follows:

```
list.add(new Student(name: "Anuj", rollNo: 2));
list.add(new Student(name: "Ramesh", rollNo: 4));
list.add(new Student(name: "Shivam", rollNo: 3));
list.add(new Student(name: "Rohit", rollNo: 1));

Student s1 = new Student(name: "Anuj", rollNo: 2);
Student s2 = new Student(name: "Rohit", rollNo: 3);

System.out.println(s1.compareTo(s2));

System.out.println(list);

Collections.sort(list, new Comparator<Student>() {
 @Override
 public int compare(Student o1, Student o2) {
 return 0;
 }
});

System.out.println(list);

List<Integer> list = new ArrayList<>();
```

The `Comparator` implementation is highlighted in blue. The output window shows the sorted list of students.

The screenshot shows the IntelliJ IDEA interface with the code editor open. The code is as follows:

```
list.add(new Student(name: "Anuj", rollNo: 2));
list.add(new Student(name: "Ramesh", rollNo: 4));
list.add(new Student(name: "Shivam", rollNo: 3));
list.add(new Student(name: "Rohit", rollNo: 1));

Student s1 = new Student(name: "Anuj", rollNo: 2);
Student s2 = new Student(name: "Rohit", rollNo: 3);

System.out.println(s1.compareTo(s2));

System.out.println(list);

Collections.sort(list, new Comparator<Student>() {
 @Override
 public int compare(Student o1, Student o2) {
 return o1.name.compareTo(o2.name);
 }
});

System.out.println(list);

List<Integer> list = new ArrayList<>();
```

The lambda expression in the `Comparator` implementation is highlighted in yellow. The output window shows the sorted list of students.

Lambda

```

 list.add(new Student(name: "Anuj", rollNo: 2));
 list.add(new Student(name: "Ramesh", rollNo: 4));
 list.add(new Student(name: "Shivam", rollNo: 3));
 list.add(new Student(name: "Rohit", rollNo: 1));

 Student s1 = new Student(name: "Anuj", rollNo: 2);
 Student s2 = new Student(name: "Rohit", rollNo: 1);

 System.out.println(s1.compareTo(s2));

 System.out.println(list);

 Collections.sort(list, (o1, o2) -> o1.name.compareTo(o2.name));

 System.out.println(list);

 List<Integer> list = new ArrayList<>();
 list.add(34);
 list.add(12);
 list.add(9);
 list.add(5);
 list.add(76);
 list.add(1);
 list.add(75);
 list.add(75);

 System.out.println("min element " + Collections.min(list));

```

**WAS THIS VIDEO HELPFUL?  
COMMENT "YES IT WAS HELPFUL"**

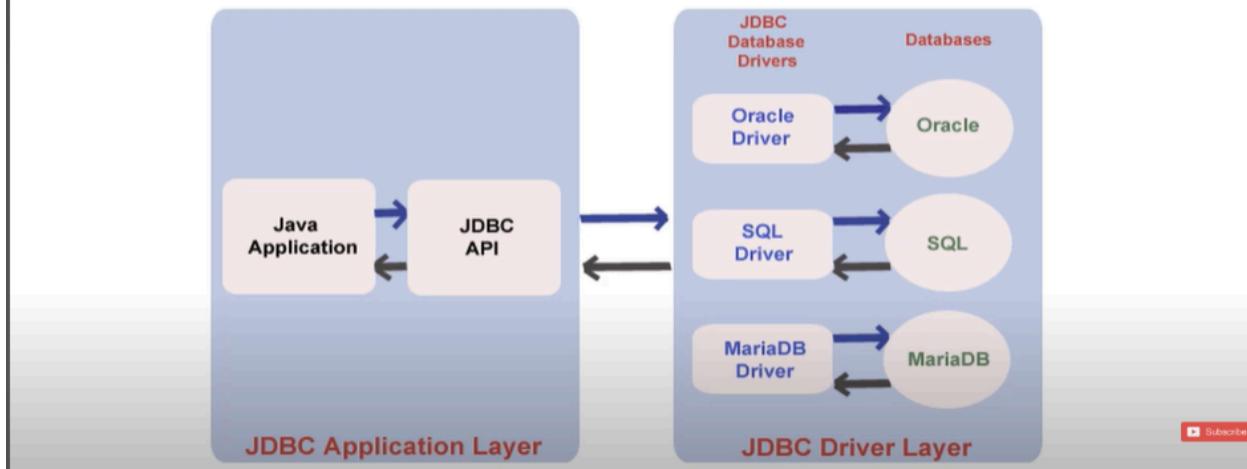
## JDBC

### JDBC-Java Database Connectivity

**JDBC** stands for **Java Database Connectivity**. It is a standard API provided by Oracle for Java applications to interact with different set of databases.



## How JDBC Work(Architecture of JDBC)



Steps to Connect with Database , explained in Detail (Java Database Connectivity) - JDBC #3  
how to connect java program with database

1) load the driver:

```
Class.forName("com.mysql.jdbc.Driver")//
```

2)Create a Connection:

```
Connection
con=DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/dbname","root","root");
```

3)Create query, Statement , PreparedStatement , CallableStatement

eg

```
String q="select * from students";
```

```
Statement stmt=con.createStatement();
```

```
ResultSet set=stmt.executeQuery(q);
```

4)Process the data :

```
while(set.next())
{
 int id=set.getInt("studentID");
 String name=set.getString("studentName");
 System.out.println(id);
 System.out.println(name);
```

}

5) Close the connection:

```
con.close();
```

Lccpc

```
1 //Program for jdbc 1 program:
2 import java.sql.*;
3 class FirstJDBC
4 {
5 public static void main(String args[])
6 {
7 try
8 {
9 //load the driver :
10 Class.forName("com.mysql.jdbc.Driver");
11
12 //creating a connection
13
14 String url="jdbc:mysql://localhost:3306/youtube";
15 String username="root";
16 String password="root";
17
18 Connection con=DriverManager.getConnection(url,username,password);
19
20 if(con.isClosed())
21 {
```

```
12 //creating a connection
13
14 String url="jdbc:mysql://localhost:3306/youtube";
15 String username="root";
16 String password="root";
17
18 Connection con=DriverManager.getConnection(url,username,password);
19
20 if(con.isClosed())
21 {
22 System.out.println("Connection is Closed");
23 }else
24 {
25 System.out.println("Connection Created...");
26 }
27 }
```

```
File Edit Selection Find View Goto Tools Project Preferences Help
java 1.8.0_101 Untitled T) used the driver: FirstJDBC.java

23 }else
24 {
25 System.out.println("Connection Created...");
26 }
27
28
29
30
31
32
33 }catch(Exception e)
34 {
35 e.printStackTrace();
36 }
37 }
38 }
39
```

Create table

```
String username="root";
String password="root";

Connection con=DriverManager.getConnection(url,username,password);

//create a query

String q="create table table1(tId int(20) primary key auto_increment, tN

//create a statement:

Statement stmt=con.createStatement();

stmt.executeUpdate(q);

System.out.println("table created in database..");

con.close();
```

Subscribe

INSERT

```
File Edit Selection Find View Goto Tools Project Preferences Help
java 1.8.0_111 11 T) load the driver InsertDBCJava InsertDBCJava RunJava
1 //PreparedStatement:insert
2
3 import java.sql.*;
4 class Fun
5 {
6 public static void main(String[] args) {
7 try
8 {
9
10 Class.forName("com.mysql.jdbc.Driver");
11
12 //creating a connection
13
14 String url="jdbc:mysql://localhost:3306/youtube";
15 String username="root";
16 String password="root";
17
18 Connection con=DriverManager.getConnection(url,username,password);
19
20
}

```

```
File Edit Selection Find View Goto Tools Project Preferences Help
java 1.8.0_111 11 T) load the driver InsertDBCJava InsertDBCJava RunJava
Connection con=DriverManager.getConnection(url,username,password);
//create a query
String q="insert into table1(tName,tCity) values (?,?)";
//get the PreparedStatement object
PreparedStatement pstmt=con.prepareStatement(q);
//set the values to query
pstmt.setString(1,"Duari");
pstmt.setString(2,"Lucknorgesh Tiww");
pstmt.executeUpdate();
System.out.println("inserted...");
```

Dynamic entry

```
String q="insert into table1(tName,tCity) values (?,?)";

//get the PreparedStatement object

PreparedStatement pstmt=con.prepareStatement(q);

BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter name :");
String name=br.readLine();

System.out.println("Enter city:");
String city=br.readLine();

//set the values to query

pstmt.setString(1,name);
pstmt.setString(2,city);

pstmt.executeUpdate();
```

## Update data in database

The screenshot shows the Eclipse IDE interface with the following details:

- File Bar:** File, Edit, View, Navigate, Source, Refactor, Run, Debug, Profile, Team, Tools, Window, Help.
- Toolbar:** Includes icons for New, Open, Save, Cut, Copy, Paste, Find, and others.
- Project Explorer (Left):** Shows a project named "JDBCTest" containing "Source Packages" (com.jdbc.practice), "Libraries" (mysql-connector.jar, JXL 1.8 Default), and "Test Libraries".
- Code Editor (Center):** Displays the Java code for "UpdateJDBC.java". The code uses JDBC to update a database table. It includes imports for BufferedReader, InputStreamReader, and Connection. A main method is shown with a try block that connects to a database using ConnectionProvider.getConnection(). It then prompts the user for new values for tName and tCity using BufferedReader.readLine() and prints them to the console.
- Navigator (Bottom Left):** Shows the "main(String[] args)" method in the code editor.

The screenshot shows a Java code editor window titled "UpdateJDBC.java". The code is a Java program that updates a database table. It uses a PreparedStatement to execute an update query. The code prompts the user to enter new values for name and city, and then updates the record where student id matches the input id.

```
14
15 * @author Durgesh
16 */
17 public class UpdateJDBC {
18
19 public static void main(String[] args) {
20 try {
21
22 Connection con=ConnectionProvider.getConnection();
23
24
25 String q="update table1 set tName=? , tCity=? where tId=?";
26
27 BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
28
29 System.out.println("Enter new name : ");
30 String name=br.readLine();
31
32
33 System.out.println("Enter new city name : ");
34 String city=br.readLine();
35
36
37 System.out.println("Enter the student id : ");
38 int id=Integer.parseInt(br.readLine());
39
40 PreparedStatement pstmt=con.prepareStatement(q);
41
42 pstmt.setString(1, name);
43 pstmt.setString(2,city);
44
45 pstmt.setInt(3, id);
```

```
public static void main(String[] args) {
 try {

 Connection con = ConnectionProvider.getConnection();

 String q = "update table1 set tName=? , tCity=? where tId=?";

 BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

 System.out.println("Enter new name : ");
 String name = br.readLine();

 System.out.println("Enter new city name : ");
 String city = br.readLine();

 System.out.println("Enter the student id : ");
 int id = Integer.parseInt(br.readLine());

 PreparedStatement pstmt = con.prepareStatement(q);

 pstmt.setString(1, name);
 pstmt.setString(2, city);

 pstmt.setInt(3, id);
 I
 pstmt.executeUpdate();

 System.out.println("done.....");
 }
}
```

## Select statement in java

```
package com.jdbc.practice;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;

public class SelectJDBC {
 public static void main(String[] args) {
 try {

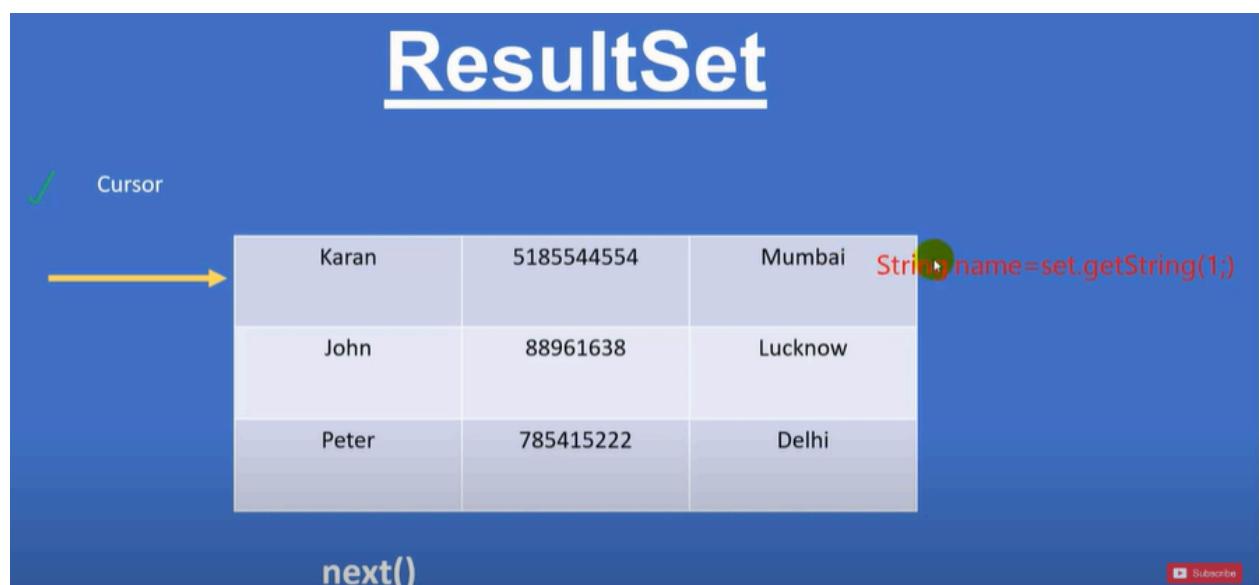
 Connection con=ConnectionProvider.getConnection();

 String query="select * from table1";

 Statement stmt=con.createStatement();
 ResultSet set=stmt.executeQuery(query);
 }
 }
}
```

```
1 package com.jdbc.practice;
2
3 import java.sql.Connection;
4 import java.sql.ResultSet;
5 import java.sql.Statement;
6
7 public class SelectJDBC {
8
9 public static void main(String[] args) {
10 try {
11
12 Connection con = ConnectionProvider.getConnection();
13
14 String query = "select * from table1";
15
16 Statement stmt = con.createStatement();
17 ResultSet set = stmt.executeQuery(query);
18
19 while (set.next()) {
20 int id = set.getInt(1);
21 String name = set.getString(2);
22 String city = set.getString(3);
23 }
24 } catch (Exception e) {
25 e.printStackTrace();
26 }
27 }
28 }
```

## Result set



## How to get object of ResultSet:

- `ResultSet set= stmt. executeQuery("select * from table" );`

**set**


Not updatable  
Cursor moves forward only



## final instance variable

---

- ❑ A java variable can be declared using the keyword **final**. Then the final variable can be assigned only once.
  
- ❑ A variable that is declared as final and not initialized is called a blank final variable. A blank final variable forces either the constructors to initialize it or initialization block to do this job.

