JAVA

OBJECT ORIENTED PROGRAMMING

Inheritance

When properties of base class are passed on to a derived class.

eg.

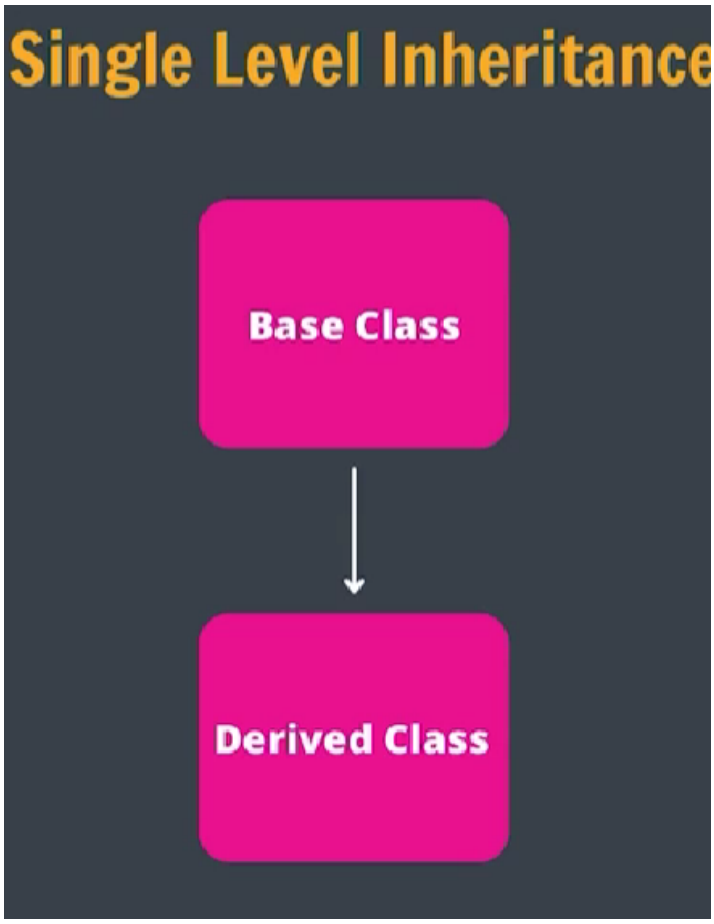
```
public class inheritance {
    public static void main(String[] args) {
        Fish shark = new Fish();
        shark.eats();
    }
}

//BASE CLASS
class Animals {
    String color ;
    void eats(){
        System.out.println("EATS");
    }
    void breathe(){
        System.out.println("BREATHES");
    }
}
//DERIVED CLASS
class Fish extends Animals {
    int fins ;
    void swim(){
        System.out.println("SWIMS IN WATER");
    }
}
```

here the keyword `extends` is used to inherit properties.

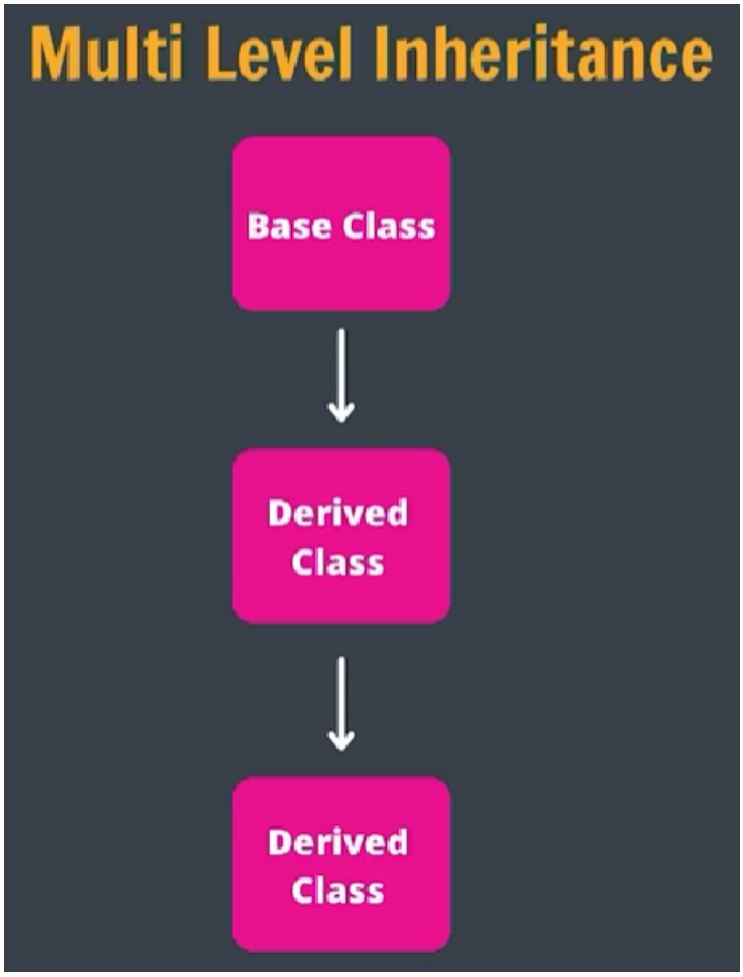
note how eat is a function of animals but it is still called from fish class.

LEVELS OF INHERITANCE

1) SINGLE LEVEL : one main class and one derived class



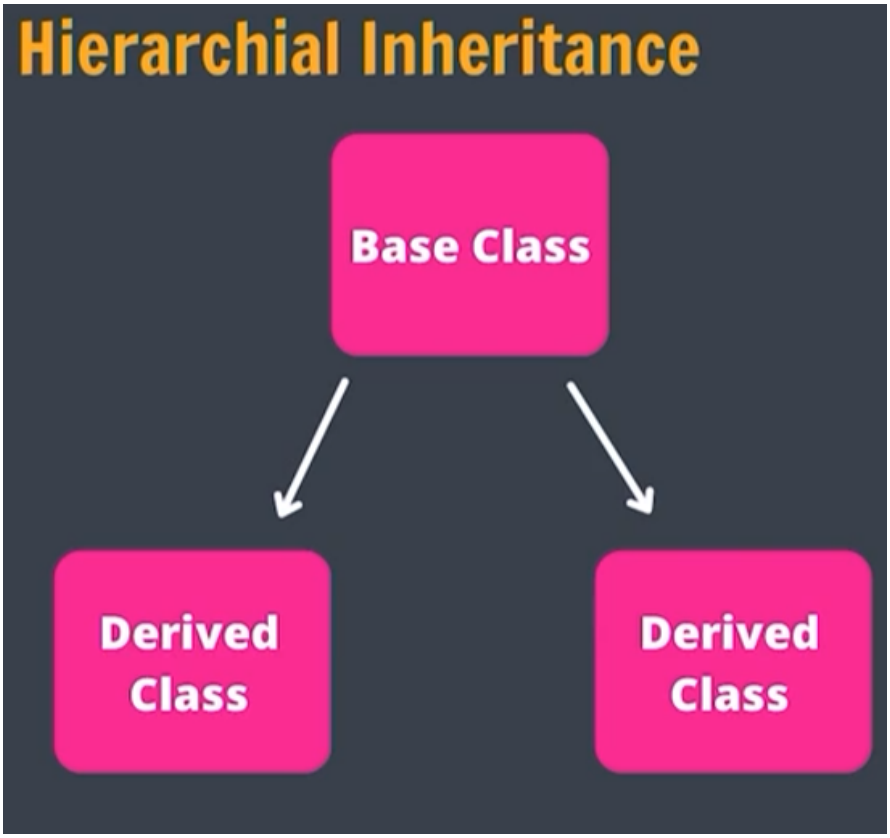
2) MULTI LEVEL :




```
public class inheritance {
    public static void main(String[] args) {
        Dogs d = new Dogs();
        d.eats();
    }
}

//BASE CLASS
class Animals {
    String color ;
    void eats(){
        System.out.println("EATS");
    }
    void breathe(){
        System.out.println("BREATHES");
    }
}
//DERIVED CLASS
class Mammal extends Animals {

    void gland(){
        System.out.println("THEY HAVE MAMMARY GLANDS");
    }
}
//DERIVED KA DERIVED CLASS
```

```
class Dogs extends Mammal {
    void pup(){
        System.out.println("THEY GIVE BIRTH TO PUPPIES");
    }
}
```

3) HIERARCHICAL INHERITANCE:



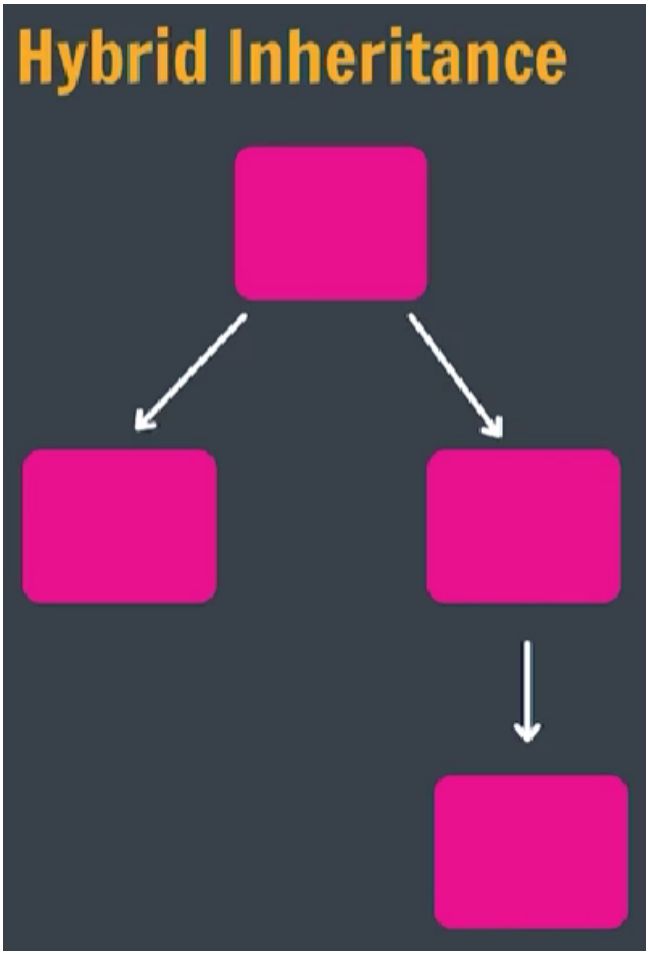

```
public class inheritance {
    public static void main(String[] args) {
        Mammal m = new Mammal();
        m.eats();
        m.walk();
        Fish f = new Fish();
        f.breathe();
        f.swim();
        Bird b = new Bird();
        b.eats();
        b.fly();
    }
}

//BASE CLASS
class Animals {
```

```
    String color ;
    void eats(){
        System.out.println("EATS");
    }
    void breathe(){
        System.out.println("BREATHES");
    }
}
//DERIVED CLASS
class Mammal extends Animals {

    void walk(){
        System.out.println("THEY walk");
    }
}
class Fish extends Animals {

    void swim(){
        System.out.println("THEY swim");
    }
}
class Bird extends Animals {
    void fly(){
        System.out.println("THEY FLY");
    }
}
```

output :

EATS
THEY walk
BREATHES
THEY swim
EATS
THEY FLY

4) HYBRID INHERITANCE :




```
public class inheritance {
    public static void main(String[] args) {
        Mammal m = new Mammal();
        m.eats();
        m.walk();
        Fish f = new Fish();
        f.breathe();
        f.swim();
        Bird b = new Bird();
        b.eats();
        b.fly();
        Humans h = new Humans();
        h.eats();
        h.smart();
    }
}

//BASE CLASS
class Animals {
    String color ;
    void eats(){
        System.out.println("EATS");
    }
    void breathe(){
        System.out.println("BREATHES");
    }
```

```
}
//DERIVED CLASS
class Mammal extends Animals {

    void walk(){
        System.out.println("THEY walk");
    }
}
//ANOTHER DERIVED CLASS
class Fish extends Animals {

    void swim(){
        System.out.println("THEY swim");
    }
}
// DERIVED CLASS OF DERIVED CLASS MAMMAL
class Humans extends Mammal {
    void smart(){
        System.out.println("THEY ARE SMART");
    }
}
//ANOTHER DERIVED CLASS
class Bird extends Animals {
    void fly(){
        System.out.println("THEY FLY");
    }
}
```

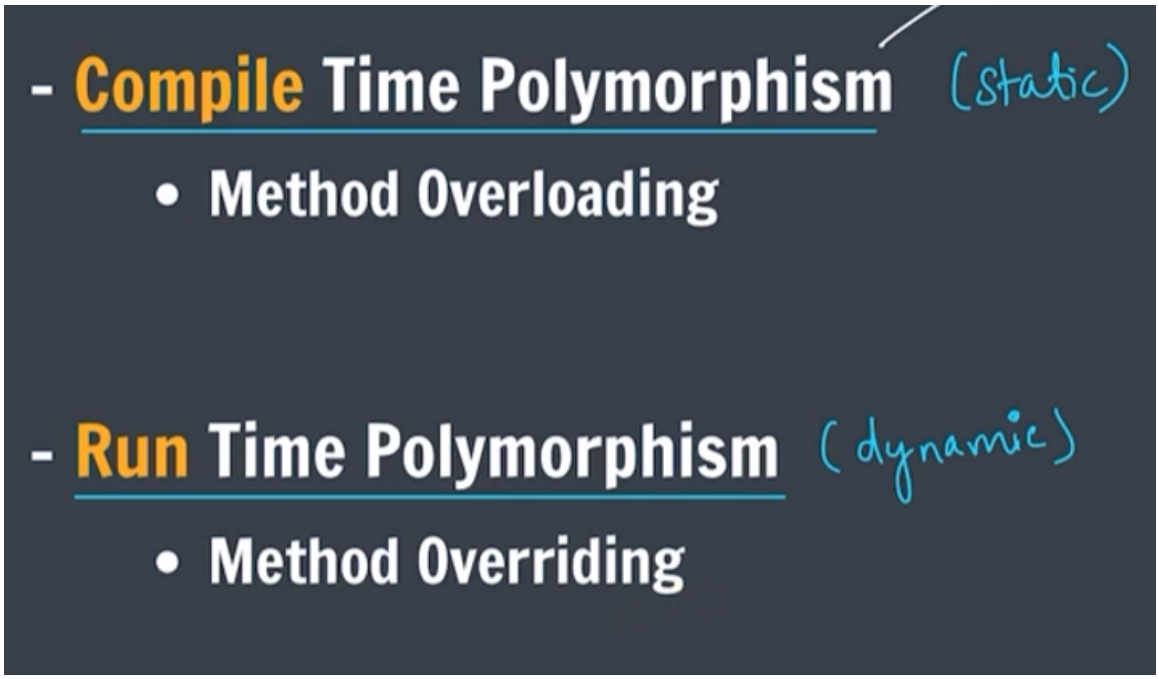
output:

EATS
THEY walk
BREATHES
THEY swim
EATS
THEY FLY
EATS
THEY ARE SMART

POLYMORPHISM

When we try to achieve similar things with multiple ways

Two types:



1) METHOD OVERLOADING : you already know

2) METHOD OVERRIDING :

```
public class inheritance {
    public static void main(String[] args) {
        Dear d = new Dear();
        d.eats();
    }
}

//BASE CLASS
class Animals {
    String color ;
    void eats(){
        System.out.println("EATS");
    }
    void breathe(){
        System.out.println("BREATHES");
    }
}
//DERIVED CLASS
class Dear {
    void eats(){
        System.out.println("EATS GRASS ONLY");
    }
}
```

output:

EATS GRASS ONLY

here when we call the derived class, so its function is called not the parents class's.

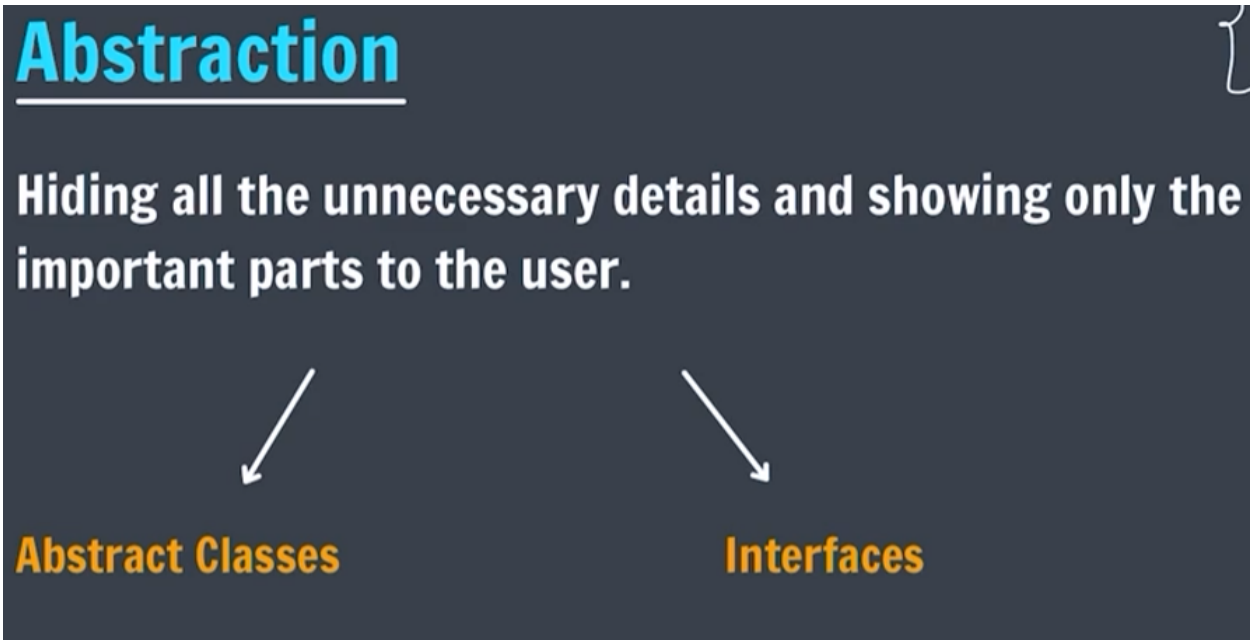
This is method overriding.

PACKAGES

Group of similar classes , sub packages and interfaces.

ABSTRACTION

Hides unimportant data and showing important data.



Abstract classes have following common properties:

1) They cannot have objects/instances.

2) Can have abstract/non abstract methods

3) Can have constructors

Abstract methods mein hum uska implimentation define nhi kr skte, they can only be defined in their derived classes for eg:

```
public class inheritance {
    public static void main(String[] args) {
        Chicken c = new Chicken();
        c.eats();
```

```
        c.walk();
        Horse h = new Horse();
        h.eats();
        h.walk();
    }
}

abstract class Animal {
    void eats(){
        System.out.println("EATS MOFO");
    }
    abstract void walk(); //NOT DEFINED CUZ ABSTRACT HAI
}
class Horse extends Animal{
    void walk(){
        System.out.println("WALKS ON 4 LEGS");
    }
}
class Chicken extends Animal{
    void walk(){
        System.out.println("WALKS ON 2 LEGS");
    }
}
```

output:

EATS MOFO
WALKS ON 2 LEGS
EATS MOFO
WALKS ON 4 LEGS

here walk is given only as idea in Animal but it is defined separately in Horse and Chicken.

Constructor initializes value for child classes.

```
public class inheritance {
    public static void main(String[] args) {
        Horse h = new Horse();
        System.out.println(h.color);
        h.changecolor();
        System.out.println(h.color);
    }
}

abstract class Animal {
    String color;
    Animal(){
        color = "Red";
    }
    void eats(){
        System.out.println("EATS MOFO");
    }
    abstract void walk(); //NOT DEFINED CUZ ABSTRACT HAI
}
class Horse extends Animal{
    void walk(){
        System.out.println("WALKS ON 4 LEGS");
    }
    void changecolor(){
        color = "Black";
```

```
    }
}
```

Output:

Red
Black

here the value of color of horse was initialized by the parent class until it was overridden by the child class function.

Always parent class ka constructor call hoga, then further derived classes ka.

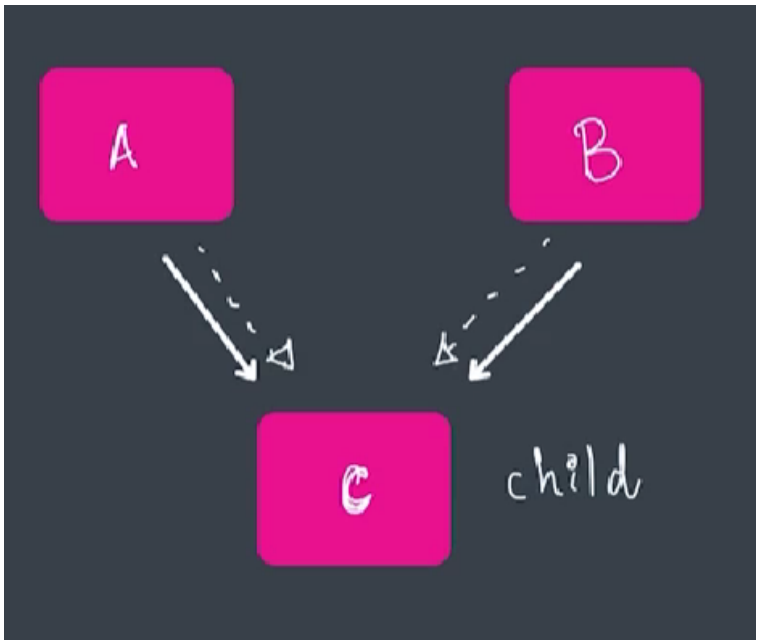
INTERFACES

They are basically next upper to classes in hierarchy.

or blueprint of class.

Two uses :

1) MULTIPLE INHERITANCE:



2) TO ACHIEVE TOTAL ABSTRACTION

NOTE: jab koi variable class k liye ek bar create hota hai, object k liye nhi , class k liye, to usse static kehdete hain

Properties of interfaces:

1) All methods are public, abstract and without implementation.

2) Used to achieve total abstraction.

3) Variables in the interfaces are final, public and static.

```
public class inheritance {
    public static void main(String[] args) {
        Queen q = new Queen();
        q.moves();
    }
}
interface ChessPlayer {
    void moves();
}
class Queen implements ChessPlayer{
    public void moves(){ //WE USE PUBLIC HERE BECAUSE
       //BY DEFAULT THE FUNCTION WOULD BE DEFINED AS "default" TYPE SO WE MAKE IT PUBLIC
        System.out.println("up,down,left,right,diagonal");
    }
}
```

Output:

up,down,left,right,diagonal

MULTIPLE INHERITANCE

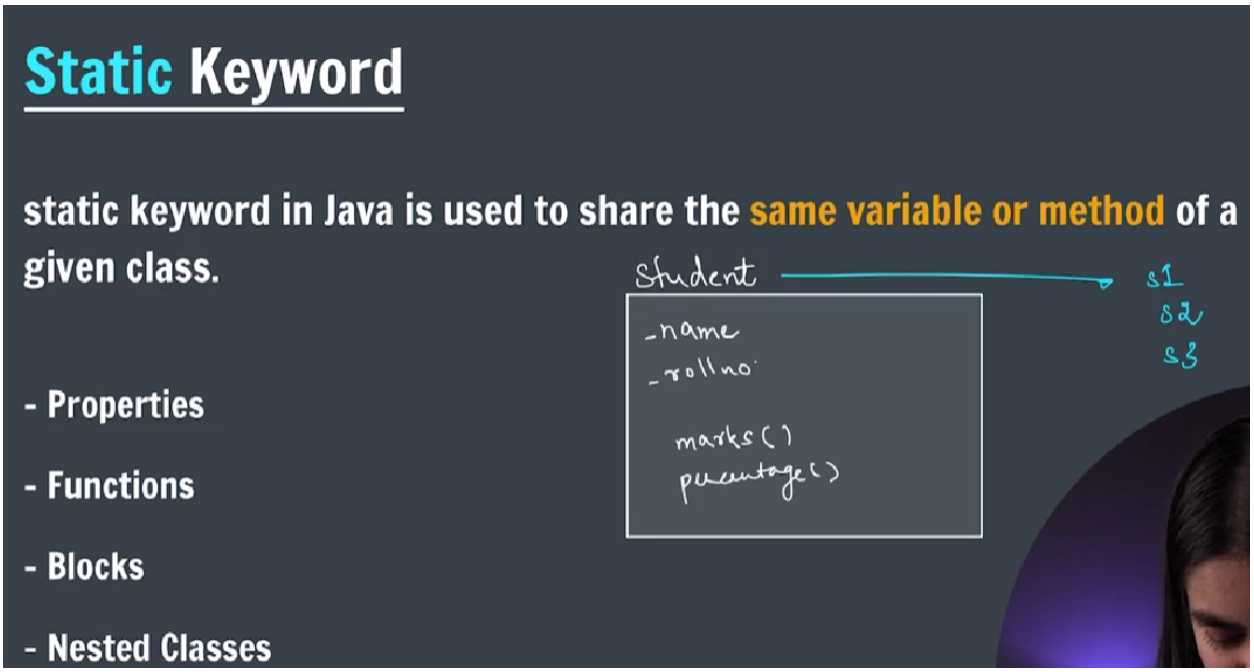
```
public class inheritance {
    public static void main(String[] args) {
        Bear b = new Bear();
        b.nonveg();
        b.vegfood();
    }
}
interface Herbivore {
    void vegfood();
}
interface Carnivore {
    void nonveg();
}
class Bear implements Carnivore, Herbivore {
    public void vegfood(){
        System.out.println("EATS VEG FOOD");
    }
    public void nonveg(){
        System.out.println("EATS NON VEG FOOD");
```

```
    }
}
```

output:

EATS NON VEG FOOD
EATS VEG FOOD

STATIC keyword



NOTE: we can define multiple classes inside a class, means nested classes

```
public class Inheritance{
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.SchoolName = "KV2";
        Student s2 = new Student();
        System.out.println(s2.SchoolName);
        Student s3 = new Student();
        s3.SchoolName = "ABC";
        System.out.println(s1.SchoolName);
    }
}
class Student {
    String name;
    int roll;
```

```
    static String SchoolName ;

    void setName(String name){
        this.name = name;
    }
    String getname(){
        return this.name;
    }
}
```

Output:

KV2

ABC

here the value of static property remains same for all until it is changed and when it is changed it becomes same for all.

When a static variable is declared, only one memory location is allocated for it, and all places where it is used , point to that memory location only.

This is the reason why we used main function as static so that only one such function exists in whole program.

SUPER keyword:

when we want to call the immediate parent constructor/class object

it has three functions:

1) access properties of parent class

2) access functions of parent class

3) access constructors of parent class

```
public class Inheritance{
    public static void main(String[] args) {
        Horse h = new Horse();
    }
}
class Animal {
    Animal(){
        System.out.println("Animal class is called");
    }
}
class Horse extends Animal {
    Horse(){
        super(); //CALLING ANIMAL CLASS
        System.out.println("Horse class is called");
    }
}
```

Output:

Animal class is called
Horse class is called

Constructor Chaining:

Constructor chaining is the process of calling one constructor from another constructor with respect to current object.

One of the main use of constructor chaining is to avoid duplicate codes while having multiple constructor (by means of constructor overloading) and make code more readable.

Constructor chaining can be done in two ways:

- **Within same class**: It can be done using `this()` keyword for constructors in the same class
- **From base class:** by using `super()` keyword to call the constructor from the base class.

Why do we need constructor chaining?

This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.

Constructor Chaining within the same class using this() keyword:

```
// Java program to illustrate Constructor Chaining
// within same class Using this() keyword
class Temp
{
  // default constructor 1
  // default constructor will call another constructor
  // using this keyword from same class
  Temp()
  {
    // calls constructor 2
    this(5);
    System.out.println("The Default constructor");
  }

  // parameterized constructor 2
  Temp(int x)
  {
    // calls constructor 3
    this(5, 15);
    System.out.println(x);
  }

  // parameterized constructor 3
  Temp(int x, int y)
  {
    System.out.println(x * y);
  }
```

```
  public static void main(String args[])
  {
    // invokes default constructor first
    new Temp();
  }
}
```

Output:

```
75
5
The Default constructor
```

Rules of constructor chaining :

1. The **this()** expression should always be the first line of the constructor.
2. There should be at-least be one constructor without the this() keyword (constructor 3 in above example).
3. Constructor chaining can be achieved in any order.

What happens if we change the order of constructors?

Nothing, Constructor chaining can be achieved in any order

Constructor Chaining to other class using super() keyword :

```
// Java program to illustrate Constructor Chaining to
// other class using super() keyword
class Base
{
  String name;

  // constructor 1
  Base()
  {
    this("");
    System.out.println("No-argument constructor of" +
                    " base class");
  }

  // constructor 2
  Base(String name)
  {
    this.name = name;
    System.out.println("Calling parameterized constructor"
                      + " of base");
  }
}

class Derived extends Base
{
  // constructor 3
  Derived()
  {
```

```
    System.out.println("No-argument constructor " +
            "of derived");
  }

  // parameterized constructor 4
  Derived(String name)
  {
    // invokes base class constructor 2
    super(name);
    System.out.println("Calling parameterized " +
            "constructor of derived");
  }

  public static void main(String args[])
  {
    // calls parameterized constructor 4
    Derived obj = new Derived("test");

    // Calls No-argument constructor
    // Derived obj = new Derived();
  }
}
```

Output:

```
Calling parameterized constructor of base
Calling parameterized constructor of derived
```

Note : Similar to constructor chaining in same class, **super()** should be the first line of the constructor as super class's constructor are invoked before the sub class's constructor.

Alternative method : using Init block:

When we want certain common resources to be executed with every constructor we can put the code in the **init block** . Init block is always executed before any constructor, whenever a constructor is used for creating a new object.

```
class Temp
{
  // block to be executed before any constructor.
  {
    System.out.println("init block");
  }

  // no-arg constructor
  Temp()
  {
    System.out.println("default");
  }

  // constructor with one argument.
  Temp(int x)
  {
    System.out.println(x);
```

```
  }

  public static void main(String[] args)
  {
    // Object creation by calling no-argument
    // constructor.
    new Temp();

    // Object creation by calling parameterized
    // constructor with one parameter.
    new Temp(10);
  }
}
```

Output:

```
init block
default
init block
10
```

NOTE: If there are more than one blocks, they are executed in the order in which they are defined within the same class.

```
class Temp
{
  // block to be executed first
  {
    System.out.println("init");
  }
  Temp()
  {
    System.out.println("default");
  }
  Temp(int x)
  {
    System.out.println(x);
  }

  // block to be executed after the first block
  // which has been defined above.
  {
    System.out.println("second");
  }
  public static void main(String args[])
  {
    new Temp();
    new Temp(10);
  }
}
```

Output :

```
init
second
default
```

```
init
second
10
```

Points to be noted:

1) Java can never have private and protected type of class as aisi class ka kaam nhi hoga kuch.

	private	default	protected	public
Class	No	Yes	No	Yes
Nested Class	Yes	Yes	Yes	Yes
Constructor	Yes	Yes	Yes	Yes
Method	Yes	Yes	Yes	Yes
Field	Yes	Yes	Yes	Yes

2) In java , child ke object ko parent class mein reference dekr bnana possible hai.

so if following happens ,

then following is possible,

left side mein reference hota hai and right side mein object create hota hai

Variable Arguments (Varargs) in Java

Variable Arguments (Varargs) in Java is a method that takes a variable number of arguments. Variable Arguments in Java simplifies the creation of methods that need to take a variable number of arguments.

Need of Java Varargs

- Until JDK 4, we cant declare a method with variable no. of arguments. If there is any change in the number of arguments, we have to declare a new method. This approach increases the length of the code and reduces readability.
- Before JDK 5, variable-length arguments could be handled in two ways. One uses an overloaded method(one for each), and another puts the arguments into an array and then passes this array to the method. Both of them are potentially error-prone and require more code.
- To resolve these problems, Variable Arguments (Varargs) were introduced in JDK 5. From JDK 5 onwards, we can declare a method with a variable number of arguments. Such types of methods are called Varargs methods. The varargs feature offers a simpler, better option.

Syntax of Varargs

Internally, the Varargs method is implemented by using the single dimensions arrays concept. Hence, in the Varargs method, we can differentiate arguments by using Index. A variable-length argument is specified by three periods or dots(…).

For Example,

```
public static void fun(int ... a)
{
   // method body
}
```

This syntax tells the compiler that fun( ) can be called with zero or more arguments. As a result, here, a is implicitly declared as an array of type int[].

```
// Java program to demonstrate varargs

class Test1 {
  // A method that takes variable
  // number of integer arguments.
  static void fun(int... a)
  {
    System.out.println("Number of arguments: "
            + a.length);

    // using for each loop to display contents of a
    for (int i : a)
      System.out.print(i + " ");
    System.out.println();
  }

  // Driver code
  public static void main(String args[])
  {
    // Calling the varargs method with
    // different number of parameters
```

```
    // one parameter
    fun(100);

    // four parameters
    fun(1, 2, 3, 4);

    // no parameter
    fun();
  }
}
```

Output

```
Number of arguments: 1
100
Number of arguments: 4
1 2 3 4
Number of arguments: 0
```

Note: A method can have variable length parameters with other parameters too, but one should ensure that there exists only one varargs parameter that should be written last in the parameter list of the method declaration. For example:

```
int nums(int a, float b, double … c)
```

In this case, the first two arguments are matched with the first two parameters, and the remaining arguments belong to c.

Erroneous Varargs Examples

Case 1: Specifying two Varargs in a single method:

```
void method(String... gfg, int... q)
{
    // Compile time error as there
    // are two varargs
}
```

Case 2: Specifying Varargs as the first parameter of the method instead of the last one:

```
void method(int... gfg, String q)
{
    // Compile time error as vararg
    // appear before normal argument
}
```

Important Points regarding Varargs

- Vararg Methods can also be overloaded, but overloading may lead to ambiguity.

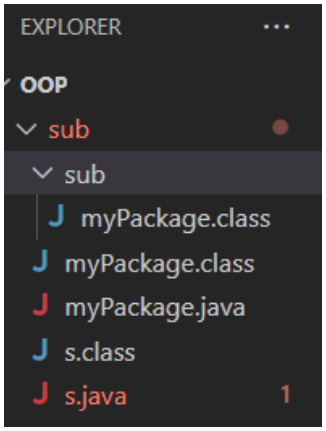
- Before JDK 5, variable length arguments could be handled in two ways: One was using overloading, other was using array argument.
- There can be only one variable argument in a method.
- Variable argument (Varargs) must be the last argument.

Creating your own package

(click above to watch video)

Create a package in a file with its classes and stuff

```
package sub;
public class myPackage {
    public static void display(){
        System.out.println("MY NAME IS ELON MUSK");
    }
    public static void main(String[] args) {
        display();
    }
}
```



create directory, and the name of subdirectory should be same as name of the package as here the name of the package is “sub” hence name of subdirectory is also sub.

then compile the package :

```
PS D:\PROGRAMMING STUFF\java programmes\OOP\sub> javac myPackage.java
```

this will create a .class file for the class

then do the following :

```
PS D:\PROGRAMMING STUFF\java programmes\OOP\sub> javac -d. myPackage.java
```

this creates a directory for all the classes of the said package

then import your package in a code file:

```
import sub.myPackage;
public class s {
    public static void main(String[] args) {
        myPackage.display();
    }
}
```

in the terminal :

```
PS D:\PROGRAMMING STUFF\java programmes\OOP\sub> javac s.java
PS D:\PROGRAMMING STUFF\java programmes\OOP\sub> java s.java
MY NAME IS ELON MUSK
```

thats how you create your own package and implement it in your programme.

EXCEPTION HANDLING IN JAVA

Java Exceptions

When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an **exception** (throw an error).

Java try and catch

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.

The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.

The `try` and `catch` keywords come in pairs:

```
try {
  //  Block of code to try
```

```
}
catch(Exception e) {
  //  Block of code to handle errors
}
```

If an error occurs, we can use `try...catch`
to catch the error and execute some code to handle it

Finally

The `finally` statement lets you execute code, after `try...catch`, regardless of the result:

```
public class Main {
  public static void main(String[] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    } finally {
      System.out.println("The 'try catch' is finished.");
    }
  }
}
```

The output will be:

`Something went wrong.`

`The 'try catch' is finished.`

self example :

```
public class exe {
    public static void main(String[] args) {
        try{
            int mynum[] = {1,2,3};
            System.out.println(mynum[10]);
        }
        catch(Exception e){
            System.out.println("something galat gaya");
        }
        finally {
            System.out.println("dekh yeh bhi execute hua");
        }
    }
}
```

Output dekh:

```
PS D:\PROGRAMMING STUFF\java programmes\OOP\sub> javac exe.java
PS D:\PROGRAMMING STUFF\java programmes\OOP\sub> java exe
```

```
something galat gaya
dekh yeh bhi execute hua
```

The throw keyword

The `throw` statement allows you to create a custom error.

The `throw` statement is used together with an **exception type**. There are many exception types available in Java: `ArithmeticException`, `FileNotFoundException`, `ArrayIndexOutOfBoundsException`, `SecurityException`, etc

```
public class exe {
    public static void main(String[] args) {
        try{
            int mynum[] = {1,2,3};
            System.out.println(mynum[10]);
        }
        catch(Exception e){
            throw new ArithmeticException("GALAT KRDIYA ");
        }
    }
}
```

Output:

```
PS D:\PROGRAMMING STUFF\java programmes\OOP\sub> java exe
Exception in thread "main" java.lang.ArithmeticException: GALAT KRDIYA
        at exe.main(exe.java:8)
```

RECURSION BASICS

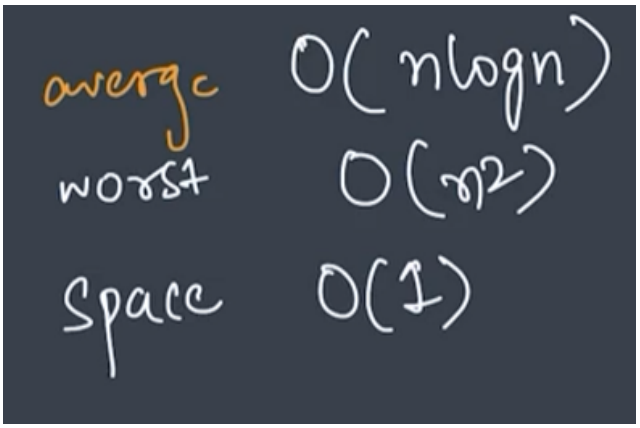
check the tiling problem, and other problems

MERGE SORT

```
public class mergesort {
    //PRINT THE ARRAY
    public static void printArray(int arr[]){
        for (int i =0 ; i<arr.length ; i++){
            System.out.print(arr[i]+" ");
        }
    }
    //DIVIDING THE ARRAY
    public static void mergeSort(int arr[], int si, int ei){
```

```
        //BASE CASE
        if(si >= ei){
            return;
        }
        int mid = si + ((ei-si)/2);
        mergeSort(arr, si, mid); //FOR LEFT SIDE
        mergeSort(arr, mid+1, ei);// FOR RIGHT SIDE
        merge(arr, si,ei,mid);//CALLING FOR MERGING
    }
    //MERGING ALL THE DIVIDED ARRAYS
    public static void merge(int arr[], int si, int ei, int mid){
        //CREATING A NEW ARRAY FOR STORING SORTED ELEMENTS
        int temp[] = new int[ei-si+1];// THIS EXPRESSION IS TO DECLARE LENGTH OF ARRAY
        int i = si;//FOR LEFT SIDE
        int j = mid+1;//FOR RIGHT SIDE
        int k = 0;//FOR TEMP WALA ARRAY

        //ALGO FOR COMPARING INDIVIDUAL ELEMENTS OF BOTH ARRAYS AND STORING THEM IN THE TEMP ARRAY AS PER CONDITION
        while(i<=mid && j<=ei){
            //IF LEFT SIDE ELEMENT IS SMALLER
            if(arr[i]<arr[j]){
                temp[k] = arr[i];
                i++; k++;
            }
            //IF RIGHT SIDE ELEMENT IS SMALLER
            else{
                temp[k] = arr[j];
                j++;k++;
            }
        }

        //FOR REMAINING ELEMENTS AFTER SORTING
        while(i<=mid){
            temp[k++]=arr[i++];
        }
        while(j<=ei){
            temp[k++]=arr[j++];
        }

        //FOR COPYING THE ELEMENTS OF TEMP ARRAY INTO THE ORIGINAL ARRAY(NOW SORTED)
        for(i=si,k=0; k<temp.length;k++,i++){
            arr[i]=temp[k];
        }
    }
    public static void main(String[] args) {
        int arr[] = {5,6,2,9,4,3,};
        mergeSort(arr, 0, arr.length-1);
        printArray(arr);
    }
}
```

QUICK SORT

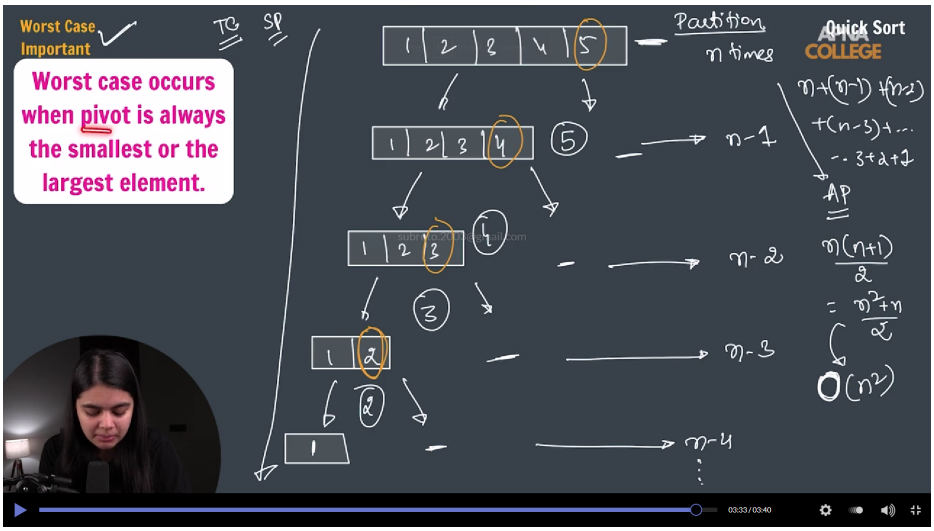



```
public class quicksort {
    //PRINTING THE ARRAY
    public static void printArray(int arr[]){
        for( int i =0; i<arr.length; i++){
            System.out.print(arr[i]+" ");
        }
    }
    //QUICK SORT, TO FIND THE PIVOT AND ALL;
    public static void quickSort(int arr[], int si, int ei){
        //BASE CASE
        if (si>=ei){
            return;
        }

        //GETTING PIVOT
        int pidx = partition(arr, si , ei);
        quickSort(arr, si, pidx-1);//FOR LEFT SIDE OF PIVOT
        quickSort(arr, pidx+1, ei);//FOR RIGHT SIDE WALA PART

    }
    //PARTITION KARKE SORT KRNE WALA FUNCTION
    public static int partition(int arr[], int si, int ei){
        int temp = 0;
        int pivot = arr[ei];
        int i = si-1;
        for(int j = si; j<=ei ; j++){
            if( arr[j] < pivot){
                i++;
                //SWAPPING VALUES
                temp = arr[j];
                arr[j] = arr[i];
                arr[i] = temp;
            }
        }
        i++;
        //SWAPPING VALUES
        temp = pivot;
        arr[ei] = arr[i];
        arr[i] = temp;
        return i ;
    }
    public static void main(String[] args) {
        int arr[] = {2,5,-3,8,9,6,3};
        quickSort(arr,0,arr.length-1);
        printArray(arr);
```

```
    }
}
```

NOTE : worst case time complexity in quick sort comes when the pivot element is always either the smallest or the largest.

in such cases it would be O(n^2).



SEARCH IN ROTATED SORTED ARRAY

```
import java.util.Scanner;

public class search {
    public static int searchInSortedArray(int arr[], int si, int ei, int target){
        //BASE CASE
        if(si > ei){
            return -1;
        }
        //find mid
        int mid = si + ((ei-si)/2);

        //MID PE HI MILJAYE AGR
        if(arr[mid] == target){
            return mid;
        }

        // LINE ONE
        if(arr[si] <= arr[mid]){
            //CASE 1: LEFT SIDE OF LINE ONE
            if(arr[si] <= target && target <= arr[mid]){
                return searchInSortedArray(arr, si, mid-1, target);
            }
            // CASE 2: RIGHT SIDE OF LINE ONE
            else{
                return searchInSortedArray(arr, mid+1, ei, target);
            }
```

```
        }
        // LINE TWO
        else{
            // CASE 3: RIGHT SIDE OF LINE 2
            if(arr[mid] <= target && target <= arr[ei] ){
                return searchInSortedArray(arr, mid+1, ei, target);
            }
            // CASE 4: LEFT SIDE OF LINE 2
            else{
                return searchInSortedArray(arr, si, mid-1, target);
            }
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int arr[] = {4,5,6,7,1,2,3};
        int target = sc.nextInt();
        int tidx = searchInSortedArray(arr,0,arr.length-1,target);
        System.out.println(tidx);
    }
}
```

explaination video: check divide and conquer folder in java programmes

Wrapper Classes in JAVA

A Wrapper class in Java is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object. Let’s check on the wrapper classes in java with examples:

Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

Below are given examples of wrapper classes in java with their corresponding Primitive data types in java.

Primitive Data types and their Corresponding Wrapper class

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Auto boxing and Unboxing - check link to continue