

# Object Oriented Programming

OOP 1 → Classes , objects , constructors and keywords.

## ① Classes and Objects

A class is a template for an object, and an object is an instance of a class.  
A class creates a new data type that can be used to create objects.

When you declare an object of a class, you are creating an instance of that class.

Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.)

## ② properties of objects

Objects are characterized by three essential properties: state, identity, and behavior.

The state of an object is a value from its data type. The identity of an object distinguishes one object from another.

It is useful to think of an object's identity as the place where its value is stored in memory.

The behavior of an object is the effect of data-type operations.

## ③ Objects and Instance

Variabhes and how to  
access them →

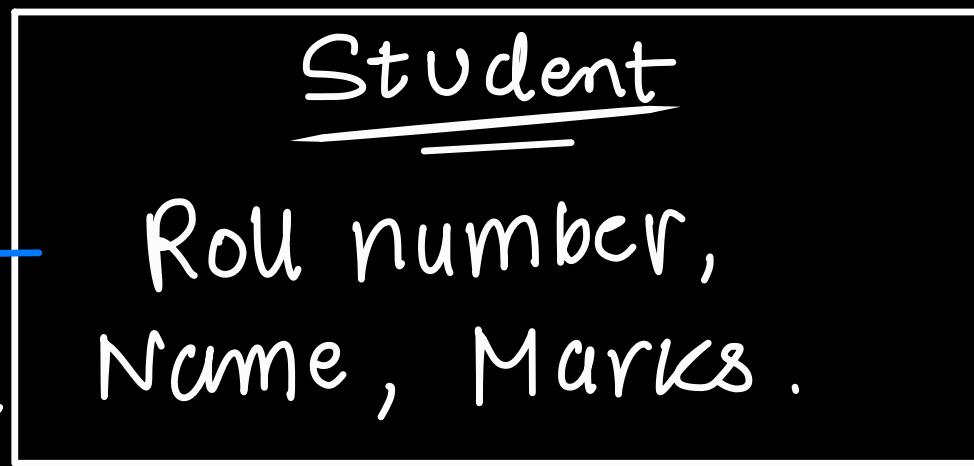
Variables  
inside the  
class.

The dot operator links the name of the object with the name of an instance variable.

Although commonly referred to as the dot operator, the formal specification for Java categorizes the . as a separator.

Inside the  
class.

Instance  
variables



Student 1



student  
2



Student  
3

Object / reference Variable

```
public class Main {
    public static void main(String[] args) {
        Student[] students = new Student[5];

        // just declaring
        Student kunal;

        System.out.println(Arrays.toString(students));
    }
}

// create a class
// for every single student
class Student {
    int rno;
    String name;
    float marks;
}
```

Printing Kunal  
will give output  
as 'null'.

→ Even printing  
array of students will  
print 5 null(s).

④ The 'new' keyword dynamically allocates(that is, allocates at run time)memory for an object & returns a reference to it.  
This reference is, more or less, the address in memory of the object allocated by new.  
This reference is then stored in the variable.  
Thus, in Java, all class objects must be dynamically allocated.

The new keyword →

Student student1; // declaring variable  
student1 = new Student();

→ dynamically allocates  
memory for an object and  
returns reference to it.

Student student1 = new Student();

↓  
Compile time

↓  
runtime

Student 2;

Student 1;

roll no.  
name  
marks

roll no.  
name  
marks

STACK

JOIN THE DARKSIDE

HEAP

# This is how dynamic Memory allocation works



It is important to understand that new allocates memory for an object during run time.

```
Box b1 = new Box();  
Box b2 = b1;
```

b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of the original object. It simply makes b2 refer to the same object as does b1. Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.

When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Declaring the object and then allocating object; why we cannot change reference in Java!

```
Box mybox; // declare reference to object  
mybox = new Box(); // allocate a Box object
```

The first line declares mybox as a reference to an object of type Box. At this point, mybox does not yet refer to an actual object. The next line allocates an object and assigns a reference to it to mybox. After the second line executes, you can use mybox as if it were a Box object. But in reality, mybox simply holds, in essence, the memory address of the actual Box object.

The key to Java's safety is that you cannot manipulate references as you can actual pointers.

Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer.

This is how to declare →

Output →

```
Student kunal = new Student();  
  
kunal.rno = 13;  
kunal.name = "Kunal Kushwaha";  
kunal.marks = 88.5f;  
  
System.out.println(kunal.rno);  
System.out.println(kunal.name);  
System.out.println(kunal.marks);  
}  
}
```

13

Kunal Kushwaha

88.5f

default value is 0, null, 0.0

Imp →

If there is already a default value set in the class itself as shown below as a code output example

Output

13

Kunal kushwahn

90.0

```
// create a class  
// for every single student  
class Student {  
    int rno;  
    String name;  
    float marks = 90;  
}
```

It will not print 88.5 as we have commented that line.

Kunal.marks = 88.5f; X

It will print the default value

## ⑤ Constructors

A Closer Look at new:

```
classname class-var = new classname ( );
```

Here, ~~class-var~~ is a variable of the class type being created. The classname is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class. A constructor defines what occurs when an object of a class is created.

Constructor is a special function.

that runs when you create an object  
and also allocates some variables

```
student student1 = new student();
```



This is a by default constructor  
even when no constructor is present  
by default-

Class w/o Constructor

```
class student {  
    int rollno;  
    String name;  
}
```

Class with constructor

```
class student {  
    int rollno;  
    String name;  
}  
student () {  
    this.rollno = 13;  
    this.name = "Lak";  
}
```

```
rno  
Main.java x  
36  
37 // create a class  
38 // for every single student  
39 class Student {  
    int rno;  
    String name;  
    float marks = 90;  
39  
40  
41  
42  
43  
44 // we need a way to add the values of the above  
// properties object by object  
45  
46 // we need one word to access every object  
47  
48  
49 Student () {  
    this.rno = 13;  
    this.name = "Kunal Kushwaha";  
    this.marks = 88.5f;  
50  
51  
52  
53  
54  
55  
56 }
```

→ This is what a constructor looks like

↓  
This keyword is replaced by Student 1, Student 2.

Example →

Student Kunal =

new Student(13, "Kunal K.", 88.5f);

This keyword in the constructor calls all the function values (r.no, name) along with the object variable.  
(Kunal in this case)

```
Student kunal = new Student();  
  
kunal.rno = 13;  
kunal.name = "Kunal Kushwaha";  
kunal.marks = 88.5f;  
  
System.out.println(kunal.rno);  
System.out.println(kunal.name);  
System.out.println(kunal.marks);  
}
```



It will become like this,

Lakshay  
is an object of  
class Student

assigning all  
the required values  
to the constructor

```
Student lakshay = new Student( roll: 13, naam: "name", perc: 99.54f);  
System.out.println(lakshay.rno);  
System.out.println(lakshay.name);  
System.out.println(lakshay.marks);
```

→ printing all

```
class Student {  
    int rno;  
    String name;  
    float marks;  
  
    Student(int roll, String naam, float perc) {  
        this.rno = roll;  
        this.name = naam;  
        this.marks = perc;  
    }  
}
```

Constructor of  
type Student  
taking values  
like a  
function

Name of these  
parameters should  
be same as present  
in the class above

These parameters  
takes values  
from  
Object declarations  
and get assigned  
to classes

⇒ If we add a function  
in our class ↓

```
Void greeting () {  
    System.out.println ("Hello" + name);  
}
```

↳ Output will be - hello Kumal

But we may use this.name  
operator.

```
Void greeting () {  
    System.out.println ("Hello" + this.name);  
}
```

↳ using this will help us in  
differentiating between diff objects

eg- Student 1 , Student 2.

(will refer to the  
current object).

The this Keyword:

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword.

this can be used inside any method to refer to the current object. That is, this  
is always a reference to the object on  
which the method was invoked.

Internally, for every object, this keyword specifies the roll no., name for every object

whenever we do  
`Lakshay.rno = 13;` → this. rno no.  
is called and  
this is replaced  
by Lakshay

## Constructor Overloading

if student kumal = new Student();  
now, as () is empty, it will go to  
the empty constructor, (which doesn't takes these  
values along with it like String  
naam, int age --- etc.)

if student kumal = new Student(13, "Kumal", 98);  
It will go to the constructor  
which takes 3 values and has 3 parameters

→ For the compilation to be successful  
each constructor must contain a different  
list of arguments.

# Calling a constructor from another constructor

Student Random = new Student();

```
}

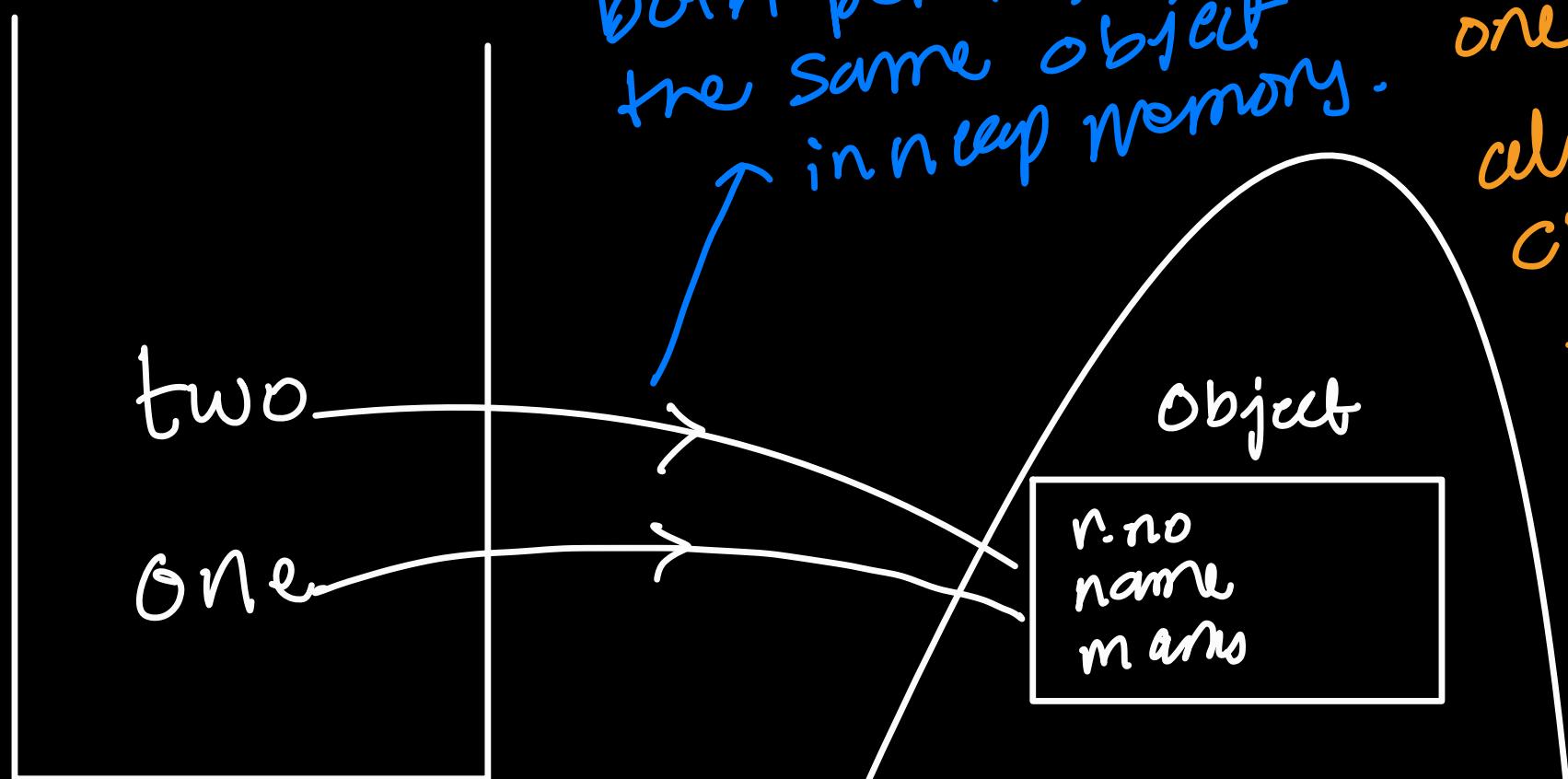
Student () {
    // this is how you call a constructor from another constructor
    // internally: new Student (13, "default person", 100.0f);
    this (rno: 13, name: "default person", marks: 100.0f);
}

// Student arpit = new Student(17, "Arpit", 89.7f);
// here, this will be replaced with arpit
Student (int rno, String name, float marks) {
    this.rno = rno;
    this.name = name;
    this.marks = marks;
}
```

← calls  
this  
constructor.



Student one = new Student();  
Student two = one;



(change in  
one will  
also  
change  
for  
two)

Stack JOIN THE DARKSIDE Heap

```
29 Constructors:  
30  
31 Once defined, the constructor is automatically called when the object is created, before the new  
operator completes.  
32 Constructors look a little strange because they have no return type, not even void.  
33 This is because the implicit return type of a class' constructor is the class type itself.  
34  
35 In the line  
36 Box mybox1 = new Box();  
37 new Box( ) is calling the Box( ) constructor.  
38  
39  
40 Inheritance and constructors in Java:  
41  
42 In Java, constructor of base class with no argument gets automatically called in derived class  
constructor.  
43 For example, output of following program given below is:  
44  
45 Base Class Constructor Called  
46 Derived Class Constructor Called  
47  
48 // filename: Main.java  
49 class Base {  
50     Base() {  
51         System.out.println("Base Class Constructor Called ");  
52     }  
53 }  
54  
55 class Derived extends Base {  
56     Derived() {  
57         System.out.println("Derived Class Constructor Called ");  
58     }  
59 }  
60  
61 public class Main {  
62     public static void main(String[] args) {  
63         Derived d = new Derived();  
64     }  
65 }  
66  
67 Any class will have a default constructor, does not matter if we declare it in the class or not. If we  
inherit a class,  
68 then the derived class must call its super class constructor. It is done by default in derived class.  
69 If it does not have a default constructor in the derived class, the JVM will invoke its default  
constructor and call  
70 the super class constructor by default. If we have a parameterised constructor in the derived class  
still it calls the  
71 default super class constructor by default. In this case, if the super class does not have a default  
constructor,  
72 instead it has a parameterised constructor, then the derived class constructor should call explicitly  
call the  
73 parameterised super class constructor.
```

## ⑥ Wrapper Classes

int a = 10; → primitive data type

Integer num = 45;

→ Wrapper Class

∴ now we can use

convert  
Int to

num. xxx = YYY; Object

↳ functions like compareTo; longValue;

## ⑦ Final Keyword

final int INCREASE = 2;



Convention is to use capital always.

Value can't be MODIFIED

As we cannot modify it ∴ always initialise the final keyword.

When final keyword is used before primitive, it cannot be modified, but when used before reference var.

Eg -

final Student Kunal = new Student();  
Kunal.name = "new name";

this is allowed

X Kunal = other object;

this is not possible, can't be  
reassigned.

```
class A {  
    final int num = 10;  
    String name;  
  
    public A(String name) {  
        this.name = name;  
    }  
}
```

final A kunal = new A(name: "Kunal Kushwaha");  
kunal.name = "other name"; → allowed

// when a non primitive is final, you cannot reassign it.  
kunal = new A(name: "new object"); → not allowed  
will give error.

```
static void swap(Integer a, Integer b) {  
    Integer temp = a;  
    a = b;  
    b = temp;  
}
```

final Keyword:

A field can be declared as final. Doing so prevents its contents from being modified, making it, essentially, a constant.

This means that you must initialize a final field when it is declared.

It is a common coding convention to choose all uppercase identifiers for final fields:

```
final int FILE_OPEN = 2;
```

Unfortunately, final guarantees immutability only when instance variables are primitive types, not reference types.

If an instance variable of a reference type has the final modifier, the value of that instance variable (the reference to an object) will never change—it will always refer to the same object—but the value of the object itself can change.

## ⑧ Garbage Collection

Java does that automatically  
we can use finalize method  
and can be called by java when  
it does garbage collection

The finalize( ) Method:

Sometimes an object will need to perform some action when it is destroyed.

To handle such situations, Java provides a mechanism called finalization. By using finalization,

you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the finalize( ) method. The Java run time calls that method whenever it is about to recycle an object of that class. Right before an asset is freed, the Java run time calls the finalize( ) method on the object.

```
protected void finalize() {  
    // finalization code here  
}
```

# Object Oriented Programming II

Packages, static, Singleton class, in-built methods

## ① Packages →

Packages are containers for classes. They are used to keep the class name space compartmentalized.

For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere. Packages are stored in a hierarchical (folder inside folder) manner and are explicitly imported into new class definitions.

it is just a folder.

com.kunal.intro → actually 3



happens when only

1 folder is present inside

the folder (just like shown  
in the IDE)

We case → same file name

e.g. - abc.java cannot be created  
in the same package ∵ it has to  
be in different packages.

Specify this for the compiler

package com.kunal.devops;

\* importing class example →

```
WrapperExample.java Greeting.java Message.java
package com.kunal.packages.a;
import static com.kunal.packages.b.Message.message; IMP

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

This is taking this  
from another class in  
the diff folder

```
WrapperExample.java Greeting.java Message.java
package com.kunal.packages.b;
public class Message {
    public static void main(String[] args) {
    }
}

public static void message() {
    System.out.println("This course is awesome");
}
```

using this  
(\* no need to use this when using insame folder)

only the ones which are  
declared as public can be accessed

cannot be accessed if declared

as private.

## More on Packages →

The package is both a naming and a visibility control mechanism.

The following statement creates a package called MyPackage: package MyPackage;

Java uses file system directories to store packages. For example, the .class files for any classes you declare to be part of MyPackage must be stored in a directory called MyPackage. Remember that case is significant, and the directory name must match the package name exactly.

A package hierarchy must be reflected in the file system of your Java development system.

For example, a package declared as

```
package java.awt.image;
```

needs to be stored in java\awt\image in a Windows environment. Be sure to choose your package names carefully.

You cannot rename a package without renaming the directory in which the classes are stored.

How does the Java run-time system know where to look for packages that you create? The answer has three parts.

- First, by default, the Java run-time system uses the current working directory as its starting point.

Thus, if your package is in a subdirectory of the current directory, it will be found.

- Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable.

- Third, you can use the -classpath option with java and javac to specify the path to your classes.

When a package is imported, only those items within the package declared as public will be available to non-subclasses in the importing code.

② Using the word **static** →

resolved  
at compile time ←

basically object of that  
class doesn't need to be created  
to access static variable

```

n.java × Main.java ×
package com.kunal.staticExample;

public class Human {
    int age;
    String name;
    int salary;
    boolean married;
    static long population;
}

public Human(int age, String name, int salary, boolean married) {
    this.age = age;
    this.name = name;
    this.salary = salary;
    this.married = married;
    Human.population += 1;
}

class Main {
    public static void main(String[] args) {
        Human kunal = new Human(age: 22, name: "Kunal", salary: 10000, married: false);
        Human rahul = new Human(age: 34, name: "Rahul", salary: 15000, married: true);

        System.out.println(kunal.population);
        System.out.println(rahul.population);
    }
}

```

is same for every object  
 ∵ declared inside the class as static.

↓  
 uses class name  
 method of (this) because same for every object.  
 we can put (this) also, but convention put class name

```

code > src > com > kunal > staticExample > Main > main
Project  Main (2)  Git: ✓ ✓ ✓
code ~/Documents/Community Class
> .idea
> out
src
  > com.kunal
    > introduction
    > packages
    > staticExample
      > Human
      > Main
        > code.html
External Libraries
Scratches and Consoles

1 package com.kunal.staticExample;
2
3 public class Main {
4     public static void main(String[] args) {
5         Human kunal = new Human(age: 22, name: "Kunal", salary: 10000, married: false);
6         Human rahul = new Human(age: 34, name: "Rahul", salary: 15000, married: true);
7
8         System.out.println(kunal.population);
9         System.out.println(rahul.population);
10    }
11 }

```

→ output shows

2 2 for both Kunal and Rahul

but if Kunal print statement + was above

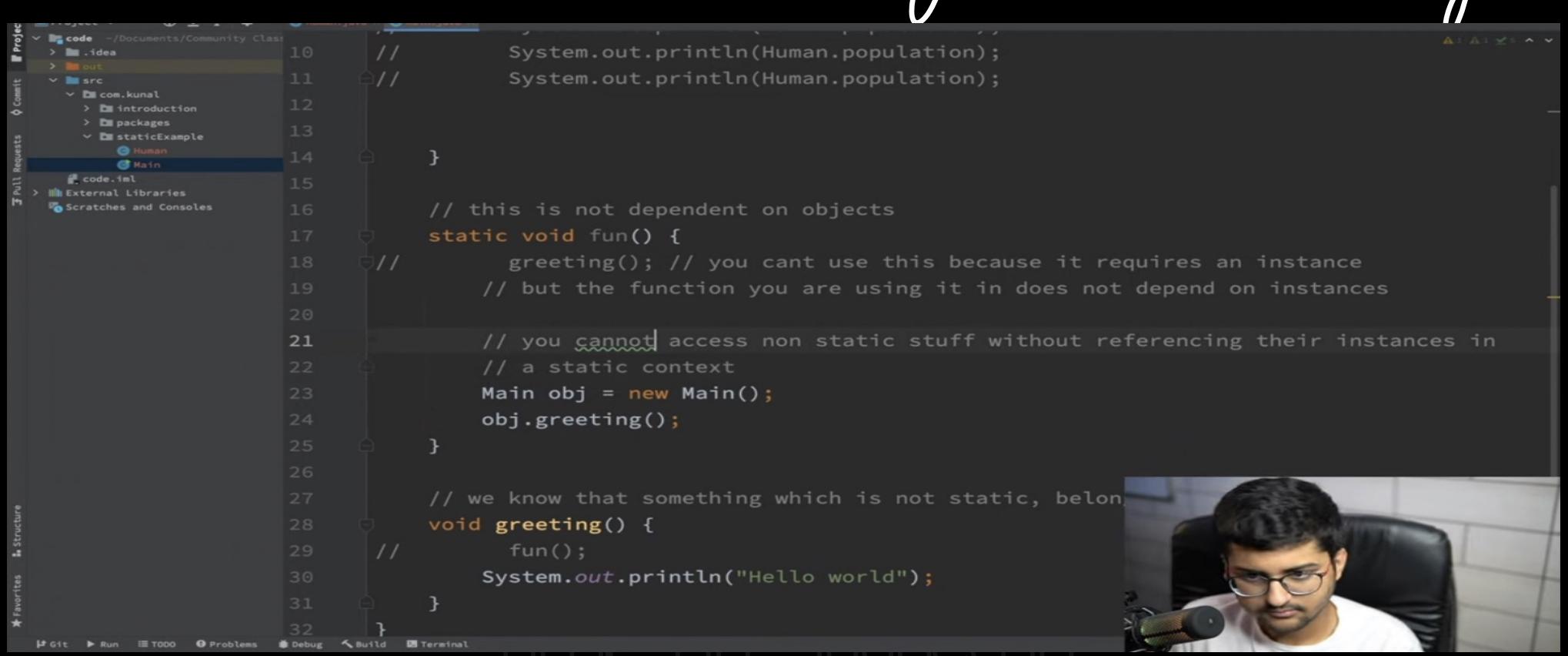
Rahul object creation : then output 1  
2.

When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be static.  
The most common example of a static member is main().  
main() is declared as static because it must be called before any objects exist.  
Static method in Java is a method which belongs to the class and not to the object.

## Static Methods →

Inside a static method (eg. main)  
we cannot use an un-static method - bcoz anything non-static belongs to an object.

Static Inside non static allowed  
We can access non static in static by referencing →



The screenshot shows a Java project structure in a code editor. The project contains a package named 'com.kunal' with classes 'Human' and 'Main'. The 'Main' class has a static void method 'fun()' that prints the population of 'Human' (which is static). It also contains a non-static void method 'greeting()' that prints 'Hello world'. A comment explains that you cannot access non-static stuff without referencing their instances in a static context. A local variable 'obj' of type 'Main' is created and its 'greeting()' method is called. The code editor's status bar at the bottom shows various icons and text.

```
// System.out.println(Human.population);
// System.out.println(Human.population);

}

// this is not dependent on objects
static void fun() {
    greeting(); // you cant use this because it requires an instance
    // but the function you are using it in does not depend on instances

    // you cannot access non static stuff without referencing their instances in
    // a static context
    Main obj = new Main();
    obj.greeting();
}

// we know that something which is not static, belon
void greeting() {
    fun();
    System.out.println("Hello world");
}
```

```

29
30 A static method can call only other static methods and cannot call a non-
31 static method from it.
32 A static method can be accessed directly by the class name and doesn't need
any object
33 A static method cannot refer to "this" or "super" keywords in anyway

```

## Static Variables →

```

1 package com.kunal.staticExample;
2
3 // this is a demo to show initialisation of static variables
4 public class StaticBlock {
5     static int a = 4;
6     static int b;
7
8     // will only run once, when the first obj is created i.e. when the class
is loaded for the first time
9     static {
10         System.out.println("I am in static block");
11         b = a * 5;
12     }           ↗ this whole static block is run only
13                               when the class
14     public static void main(String[] args) {           is first loaded.
15         StaticBlock obj = new StaticBlock();
16         System.out.println(StaticBlock.a + " " + StaticBlock.b);
17     }           ↙ loads
18     StaticBlock.b += 3;
19
20     System.out.println(StaticBlock.a + " " + StaticBlock.b);
21
22     StaticBlock obj2 = new StaticBlock();
23     System.out.println(StaticBlock.a + " " + StaticBlock.b);
24 }           ↙ again loads the class
25           but static will only run once (1st time) → output
26           ↙

```

If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded.

```

// Demonstrate static variables, methods, and blocks.
class UseStatic {
    static int a = 3;
    static int b;
    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
    static {
        System.out.println("Static block initialized.");
        b = a * 4;
    }
    public static void main(String args[]) {
        meth(42);
    }
}

```

As soon as the UseStatic class is loaded, all of the static statements are run. First, a is set to 3, then the static block executes, which prints a message and then initializes b to  $a^4$  or 12. Then main( ) is called, which calls meth( ), passing 42 to x. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

Here is the output of the program:

Static block initialized. x = 42

a = 3

b = 12

Note: main method is static, since it must be accessible for an application to run, before any instantiation takes place.

## Inner Class [Class inside a Class]

\* outside classes cannot be static  
only inner can be static. because  
itself not dependent on any class .

```
package com.kunal.staticExample;

public class InnerClasses {

    class Test {
        String name;

        public Test(String name) {
            this.name = name;
        }
    }

    public static void main(String[] args) {
        Test a = new Test(name: "Kunal");
        Test b = new Test(name: "Rahul");
    }
}
```

gives an error  
Test is a nonstatic class

```
package com.kunal.staticExample;

public class InnerClasses {

    static class Test {
        String name;
        public Test(String name) {
            this.name = name;
        }
    }

    public static void main(String[] args) {
        Test a = new Test(name: "Kunal");
        Test b = new Test(name: "Rahul");
    }
}

// static class A
// // //
```

wrks because  
class is static  
now .

object of  
that class no need to run any outside class  
this. not dependent on

also in a static variable, to <sup>upper</sup>  
access it we use → class-variable.

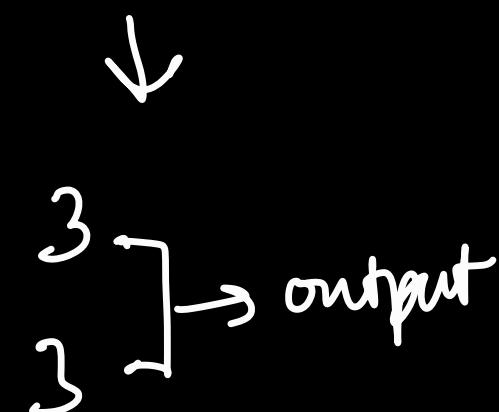
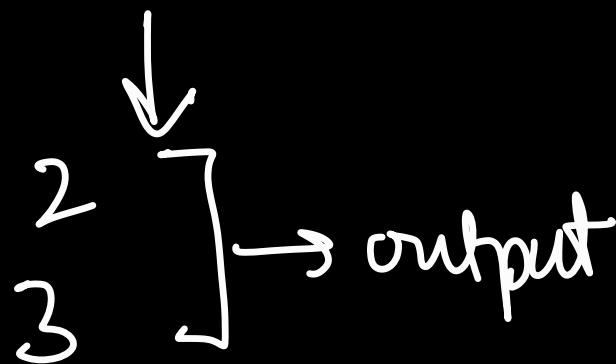
not → this-variable



↓ but

↙ ↘ ∴ a.variable = 2;  
↙ ↗ b.variable = 3;

class.variable = 2;  
class.variable = 3;



```
37 // Demonstrate static variables, methods, and blocks.  
38 class UseStatic {  
39     static int a = 3;  
40     static int b;  
41     static void meth(int x) {  
42         System.out.println("x = " + x);  
43         System.out.println("a = " + a);  
44         System.out.println("b = " + b);  
45     }  
46     static {  
47         System.out.println("Static block initialized.");  
48         b = a * 4;  
49     }  
50     public static void main(String args[]) {  
51         meth(42);  
52     }  
53 }
```

55 As soon as the UseStatic class is loaded, all of the static statements are run. First, a is set to 3,  
56 then the static block executes, which prints a message and then initializes b to a\*4 or 12. Then  
main( ) is called,  
57 which calls meth( ), passing 42 to x. The three println( ) statements refer to the two static  
variables a and b,  
58 as well as to the local variable x.

60 Here is the output of the program:

```
61 Static block initialized. x = 42  
62 a = 3  
63 b = 12
```

64 Note: main method is static, since it must be accessible for an application to run, before any  
instantiation takes place.

```
65  
66 NOTE: Only nested classes can be static.  
67 NOTE: Static inner classes can have static variables
```

```

70 run-time and not compile time, and then calling the respective method.
71 Static methods are class level methods, so it is always resolved during compile time.
72 Static INTERFACE METHODS are not inherited by either an implementing class or a sub-interface.
73
74 NOTE:
75 public class Static {
76
77     // class Test // ERROR
78     static class Test{
79         String name;
80
81         public Test(String name) {
82             this.name = name;
83         }
84     }
85
86     public static void main(String[] args) {
87         Test a = new Test("Kunal");
88
89         Test b = new Test("Rahul");
90
91         System.out.println(a.name); // Kunal
92         System.out.println(b.name); // Rahul
93     }
94 }
95 Because :
96 The static keyword may modify the declaration of a member type C within the body of a non-inner class
or interface T.
97 Its effect is to declare that C is not an inner class. Just as a static method of T has no current
instance of T in its
98 body, C also has no current instance of T, nor does it have any lexically enclosing instances.
99 Here, test does not have any instance of its outer class Static. Neither does main.
100 But main & Test can have instances of each other.

```

## Singleton Class →

only 1 object allowed from the class-

```

package com.kunal.singleton;

import com.kunal.access.A;

public class Singleton {
    private Singleton () {
        → This is a constructor
    }

    private static Singleton instance; → Object

    public static Singleton getInstance() {
        // check whether 1 obj only is created or not
        if (instance == null) {
            instance = new Singleton();
        }
        → doesn't allow a diff
        return instance;
    }
}

```

```
package com.kunal.singleton;
```

## Main Method

```
import com.kunal.access.A;
```

```
public class Main {
```

```
    public static void main(String[] args) {  
        Singleton obj1 = Singleton.getInstance();
```

```
        Singleton obj2 = Singleton.getInstance();
```

```
        Singleton obj3 = Singleton.getInstance();
```

```
        // all 3 ref variables are pointing to just one object
```

```
A a = new A(10, "Kunal");
```

```
a.getNum();
```

```
//
```

```
    int n = a.num,
```

```
}
```

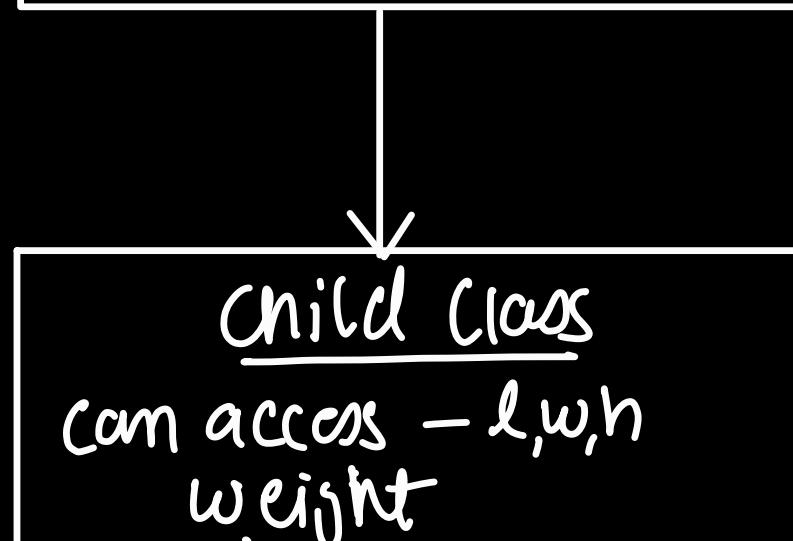
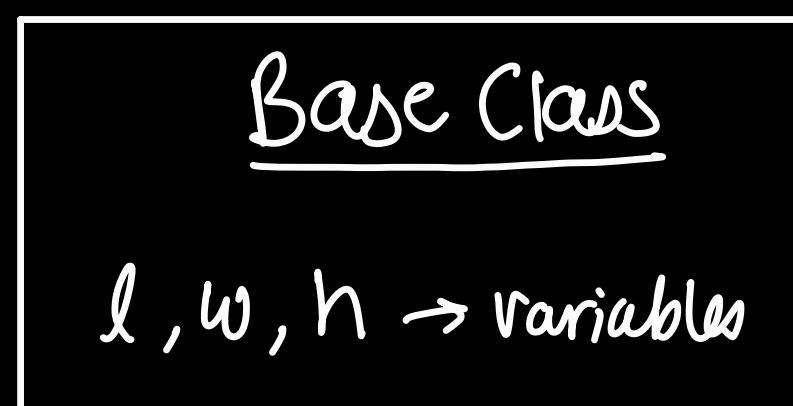
not allowed only points  
to obj of obj1

# Object Oriented Programming III

Principles - Inheritance, Polymorphism, Encapsulation, Abstraction.

## I Inheritance

child class will have properties of its own + it can access the properties (variables) of the base class.



child's own variable

Child class is inheriting properties from the base class.

Class Child extends Base {  
    int weight;

}

Child baby = new Child();  
    baby.l;  
    baby.w;  
    baby.h;

}     can access all these elements

1 To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword.

2  
3 class subclass-name extends superclass-name { // body of class  
4 }

5 You can only specify one superclass for any subclass that you create. Java does not support the inheritance of  
6 multiple superclasses into a single subclass. You can, as stated, create a hierarchy of inheritance in  
7 which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

8  
9 Although a subclass includes all of the members of its superclass, it cannot access those members of  
10 the superclass  
11 that have been declared as private.

# how it works →

```
package com.kunal.properties.inheritance;

public class BoxWeight extends Box{
    double weight;
    public BoxWeight() {
        this.weight = -1;
    }
    public BoxWeight(double l, double h, double w, double weight) {
        super(l, h, w); // what is this? call the parent class constructor
        // used to initialise values present in parent class
        this.weight = weight;
    }
}
```

BoxWeight.java

extends key word - inherits the Box class  
a simple constructor  
always important to initialise this super(l,h,w) first  
↳ This constructor takes 4 values, 3 using super key word

```
BoxWeight.java * Main.java *

package com.kunal.properties.inheritance;

public class Main {
    public static void main(String[] args) {
        Box box1 = new Box(4.6, 7.9, 9.9);
        Box box2 = new Box(box1);
        System.out.println(box1.l + " " + box1.w + " " + box1.h);

        BoxWeight box3 = new BoxWeight();
        BoxWeight box4 = new BoxWeight(l:2, h:3, w:4, weight:8);
        System.out.println(box3.h + " " + box3.weight);
    }
}
```

Main.java

↳ calls and prints

```
Box.java

1 package com.kunal.properties.inheritance;
2
3 public class Box {
4     private double l,
5     double h;
6     double w;
7     //     double weight;
8
9     static void greeting() {
10         System.out.println("Hey, I am in Box class. Greetings!");
11     }
12
13     public double getL() {
14         return l;
15     }
16
17     Box () {
18         this.h = -1;
19         this.l = -1;
20         this.w = -1;
21     }
22
23     // cube
24     Box (double side) {
25         // super(); Object class
26         this.w = side;
27         this.l = side;
28         this.h = side;
29     }
30
31     Box(double l, double h, double w) {
32         System.out.println("Box class constructor");
33         this.l = l;
34         this.h = h;
35         this.w = w;
36     }
}
```

Box.java

super(l,h,w) calls this constructor  
↳ if a variable is declared as private, it can't be used by the child class.

base class for the  
→ Box class, base class for the  
child class, BoxWeight  
↳ various constructors

K S I D E

```

13
14
15 Box box5 = new BoxWeight(2, 3, 4, 8);
16 System.out.println(box5.w);
17
18 // there are many variables in both parent and child classes
19 // you are given access to variables that are in the ref type i.e. BoxWeight
20 // hence, you should have access to weight variable
21 // this also means, that the ones you are trying to access should be initialised
22 // but here, when the obj itself is of type parent class, how will you call the constructor of child class
23 // this is why error
24 BoxWeight box6 = new Box(2, 3, 4);
25 System.out.println(box6);
26

```

This can access l, w, h  
but not weight, even when wt is being  
initialised → This is because this defines reference variable  
not the new boxwt();

can't access (l, w, h)  
because they were never  
initialised. as BoxWeight is the  
defining one

```

11
12 A Superclass Variable Can Reference a Subclass Object:
13 It is important to understand that it is the type of the reference variable—not the type of the object
that it refers
14 to—that determines what members can be accessed.
15 When a reference to a subclass object is assigned to a superclass reference variable, you will have
access only to
16 those parts of the object defined by the superclass.
17
18 plainbox      = weightbox;
19 (superclass)   (subclass)
20
21 SUPERCLASS ref = new SUBCLASS();    // HERE ref can only access methods which are available in
SUPERCLASS
22

```

## Uses of Super keyword →

Using super:

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super.

super has two general forms. The first calls the superclass' constructor.

The second is used to access a member of the superclass that has been hidden by a member of a subclass.

```

23
24
25 BoxWeight(double w, double h, double d, double m) {
26     super(w, h, d); // call superclass constructor
27     weight = m;
28 }
29
30 Here, BoxWeight( ) calls super( ) with the arguments w, h, and d. This causes the Box constructor to
31 be called,
32 which initializes width, height, and depth using these values. BoxWeight no longer initializes these
33 values itself.
34 It only needs to initialize the value unique to it: weight. This leaves Box free to make these values
35 private if desired.
36
37 Thus, super( ) always refers to the superclass immediately above the calling class.
38 This is true even in a multileveled hierarchy.
39
40 class Box {
41     private double width;
42     private double height;
43     private double depth;
44
45     // construct clone of an object
46

```

```

47 Box(Box ob) { // pass object to constructor
48     width = ob.width;
49     height = ob.height;
50     depth = ob.depth;
51 }
52 }

53
54 class BoxWeight extends Box {
55     double weight; // weight of box
56
57     // construct clone of an object
58
59     BoxWeight(BoxWeight ob) { // pass object to constructor
60         super(ob);
61         weight = ob.weight;
62     }
63 }
64

65 Notice that super() is passed an object of type BoxWeight—not of type Box. This still invokes the
constructor Box(Box ob).
66 NOTE: A superclass variable can be used to reference any object derived from that class.
67 Thus, we are able to pass a BoxWeight object to the Box constructor. Of course, Box only has knowledge
of its own members.
68

```

exactly  
same thing  
happening

(2)

```

69 A Second Use for super
70 The second form of super acts somewhat like this, except that it always refers to the superclass of
the subclass in
71 which it is used.
72
73 super.member
74
75 Here, member can be either a method or an instance variable. This second form of super is most
applicable to situations
76 in which member names of a subclass hide members by the same name in the superclass.
77

```

example int weight is in  
both Box and BoxWeight  
so Super.weight → access to Box weight  
this.weight → access to BoxWeight  
variable

\* Imp

```

78 super( ) always refers to the constructor in the closest superclass. The super( ) in BoxPrice calls
the constructor in
79 BoxWeight. The super( ) in BoxWeight calls the constructor in Box. In a class hierarchy, if a
superclass constructor
80 requires parameters, then all subclasses must pass those parameters "up the line." This is true
whether or not a
81 subclass needs parameters of its own.
82
83 If you think about it, it makes sense that constructors complete their execution in order of
derivation.
84 Because a superclass has no knowledge of any subclass, any initialization it needs to perform is
separate from and
85 possibly prerequisite to any initialization performed by the subclass. Therefore, it must complete its
execution first.
86
87 NOTE: If super( ) is not used in subclass' constructor, then the default or parameterless constructor
of each superclass
88 will be executed.
89

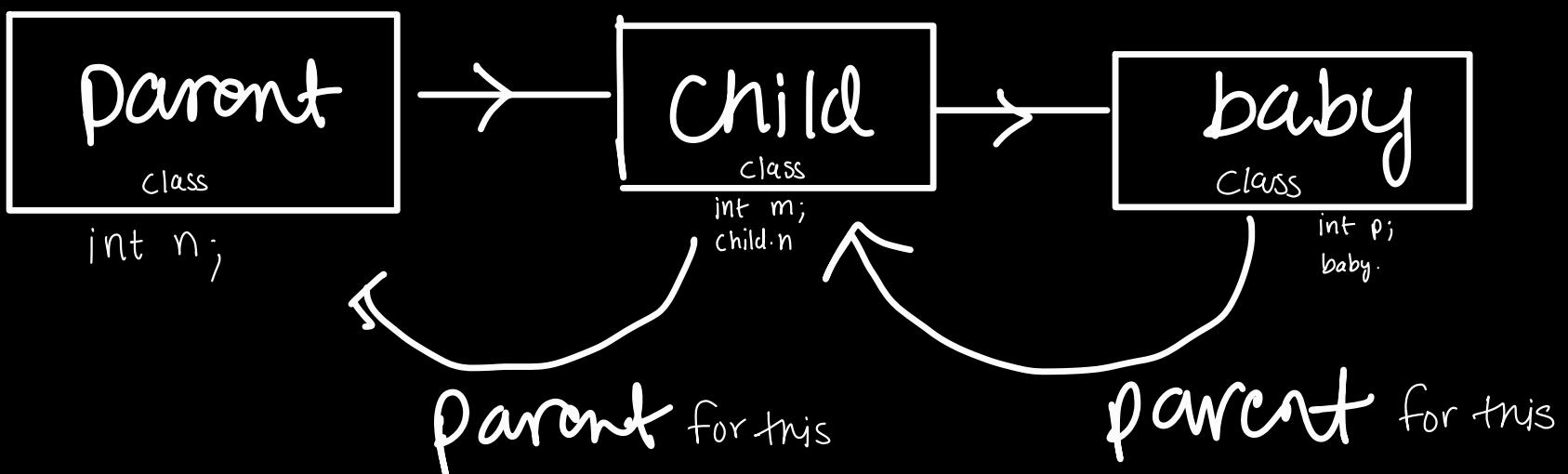
```

# Types of Inheritance

## ① single



## ② multilevel - child class will become parent class for another baby class

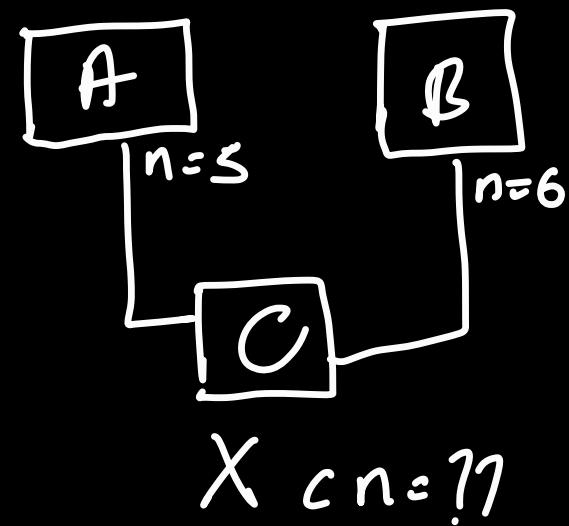


baby can use variables of parent, but super keyword in baby will refer to just 1' level up i.e Child class

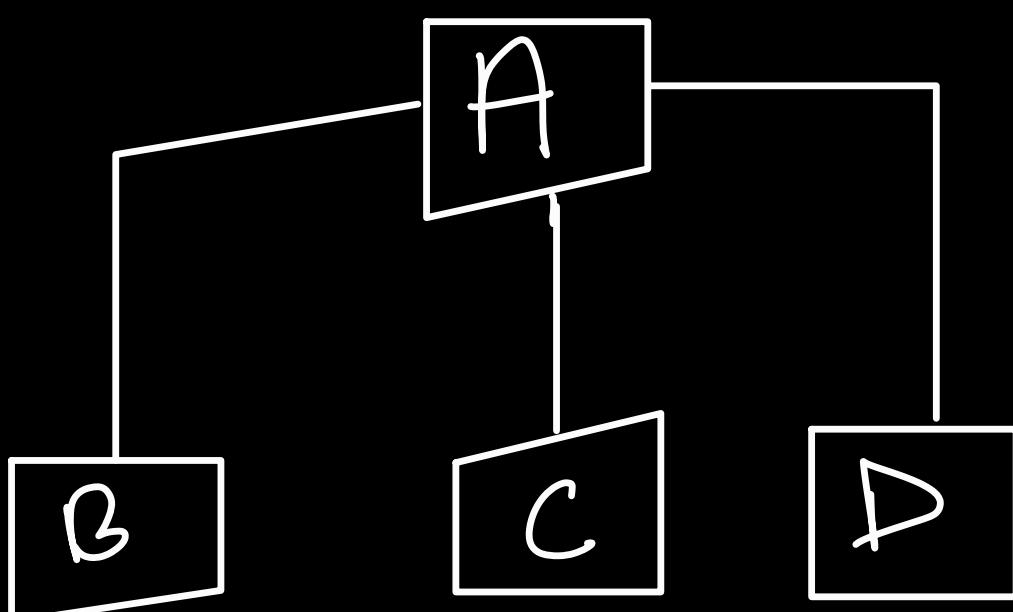
## ③ multiple inheritance

X not supported in Java

We use interfaces.



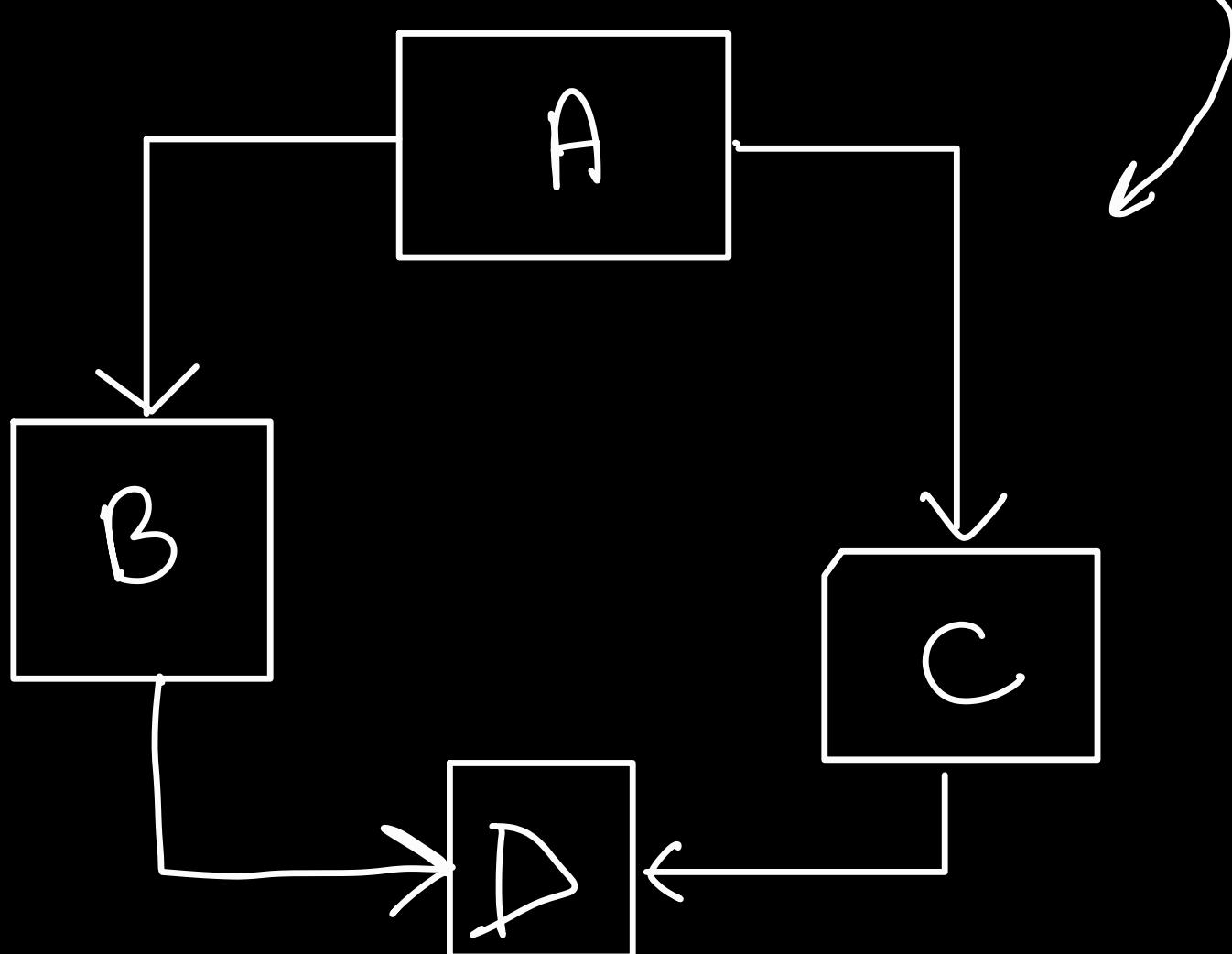
## ④ heirarchical Inheritance



B, C, D can always extend A. B can't extend C.

## ⑤ Hybrid inheritance (not in Java)

combination of single and multiple.



**Imp** - A class cannot be its own super class.

Polymorphism → many ways to represent properties of OOPs.

## ① Compile time / static Polymorphism

Achieved via method overloading



Same name of the methods but type, argument, return type, ordering can be different.  
eg - multiple constructors

Class a = new Class();

class b = new class(3,4); ] → diff constructor  
↳ Compile time polymorphism  
by method overloading

Java determines which one to run at compile time.

```
public class Numbers {  
    ① int sum(int a, int b) {  
        return a + b;  
    }  
  
    ② int sum(int a, int b, int c) {  
        return a + b + c;  
    }  
  
    public static void main(String[] args) {  
        Numbers obj = new Numbers();  
        obj.sum(a: 2, b: 3); → 2 parameters ∴ calls ①  
        obj.sum(a: 1, b: 3, c: 7); → 3 parameters ∴ calls ②  
        obj.sum(4, 5, 6, 8);  
    }  
}
```

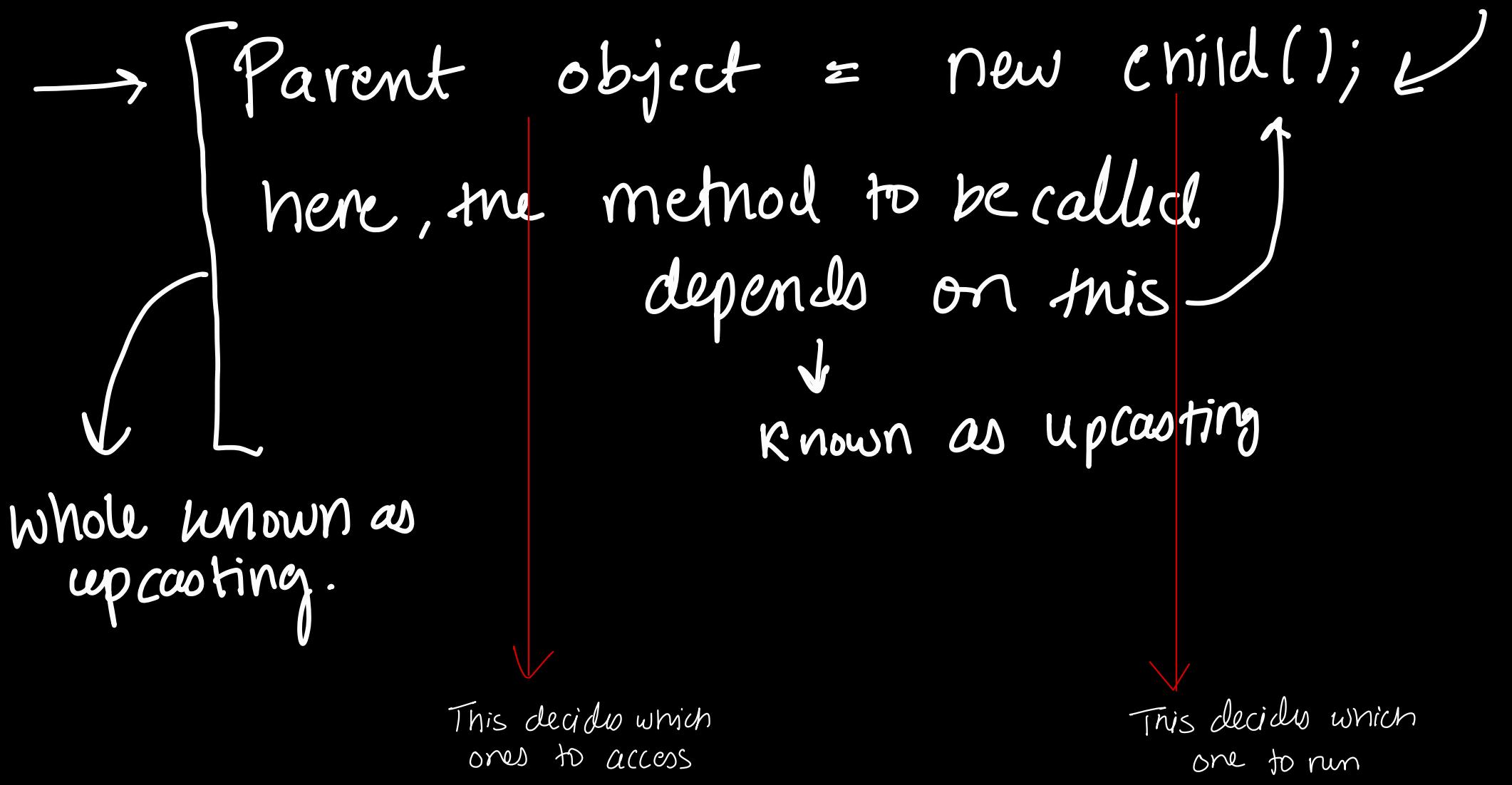
Both these methods have the same name

↳ error as checking at compile time

\* - if double in parameter and we give argument as int, Auto Casting happens.

## ② Runtime / Dynamic Polymorphism

Parent has function called Area  
Child also has function called Area  
when object. Area is called ↴



how Java does this → at compile time it decides which methods it can access (∴ here from parent class)  
 so method (Area) should be there in parent class and at compile time, the subclass's method is run, because it decides which one to run

```

1 In a class hierarchy, when a method in a subclass has the same name and type signature as a method in
2 its superclass,
3 then the method in the subclass is said to override the method in the superclass. When an overridden
4 method is called
5 from within its subclass, it will always refer to the version of that method defined by the subclass.
6 The version of the
7 method defined by the superclass will be hidden.
8
9 Method overriding occurs only when the names and the type signatures of the two methods are identical.
10 If they are not, then the two methods are simply overloaded.
11
12 (Check display functions in box classes)
13
14 Dynamic Method Dispatch:
15
16 Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run
17 time, rather than
18 compile time. Dynamic method dispatch is important because this is how Java implements run-time
19 polymorphism.
20 Let's begin by restating an important principle: a superclass reference variable can refer to a
21 subclass object.
22 When an overridden method is called through a superclass reference, Java determines which version of
23 that method to
24 execute based upon the type of the object being referred to at the time the call occurs. Thus, this
25 determination is
26 made at run time.
27 In other words, it is the type of the object being referred to (not the type of the reference variable)
28 that determines which version of an overridden method will be executed.
29
30 If B extends A then you can override a method in A through B with changing the return type of method to
31 B.
  
```

→ A  
 better explanation.

```

1 package com.kunal.properties.polymorphism;
2
3 public class Main {
4     public static void main(String[] args) {
5         Shapes shape = new Shapes();
6         Shapes circle = new Circle();
7         Shapes square = new Square();
8
9         circle.area();
10    }
11 }

```

```

1 package com.kunal.properties.polymorphism;
2
3 public class Shapes {
4     void area() {
5         System.out.println("I am in shapes");
6     }
7 }

```

This method calls the  
One in area , not in  
super class .

This one is called  
at runtime instead  
of this

```

1 package com.kunal.properties.polymorphism;
2
3 public class Circle extends Shapes{
4
5     // this will run when obj of Circle is created
6     // hence it is overriding the parent method
7     @Override // this is called annotation
8     void area() {
9         System.out.println("Area is pi * r * r");
10    }
11 }

```

Final Keyword to prevent Overriding →

```

97 # Using final to Prevent Overriding:
98 To disallow a method from being overridden, specify final as a modifier at the start of its
declaration.
99 Methods declared as final cannot be overridden.
100 Methods declared as final can sometimes provide a performance enhancement: The compiler is free to
inline calls to them
101 because it "knows" they will not be overridden by a subclass. When a small final method is called,
often the Java
102 compiler can copy the bytecode for the subroutine directly inline with the compiled code of the
calling method, thus
103 eliminating the costly overhead associated with a method call. Inlining is an option only with final
methods.
104 Normally, Java resolves calls to methods dynamically, at run time. This is called late binding.
However, since final
105 methods cannot be overridden, a call to one can be resolved at compile time. This is called early
binding.
106

```

# Final Keyword to prevent Inheritance →

```
107 # Using final to Prevent Inheritance:  
108 Sometimes you will want to prevent a class from being inherited. To do this, precede the class  
declaration with final.  
109 NOTE: Declaring a class as final implicitly declares all of its methods as final, too.  
110 As you might expect, it is illegal to declare a class as both abstract and final since an abstract  
class is incomplete  
111 by itself & relies upon its subclasses to provide complete implementations.  
112  
113 # NOTE: Although static methods can be inherited ,there is no point in overriding them in child  
classes because the  
114 method in parent class will run always no matter from which object you call it. That is why static  
interface methods  
115 cannot be inherited because these method will run from the parent interface and no matter if we were  
allowed to  
116 override them, they will always run the method in parent interface.  
117 That is why static interface method must have a body.  
118  
119 NOTE : Polymorphism does not apply to instance variables.  
120
```

Static always run  
∴ one in parent class will run and the same method in child class will not be overridden

∴ overriding happens from objects  
∴ static doesn't deal with objects ∴

↑  
overriding static methods

## ③ Encapsulation

Wrapping up the implementation of the data members and methods in a class.

## ④ Abstraction

Hiding unnecessary details and showing valuable information only.

Abstraction — The design part

Encapsulation — The implementation of designed part

## Abstraction

## Encapsulation

① Abstraction is a feature of OOPs that hides the <b>unnecessary</b> detail but shows the essential information.	Encapsulation is also a feature of OOPs. It hides the code and data into a <b>single</b> entity or unit so that the data can be protected from the outside world.
② It solves an issue at the <b>design</b> level.	Encapsulation solves an issue at <b>implementation</b> level.
③ It focuses on the <b>external</b> lookout.	It focuses on <b>internal</b> working.
④ It can be implemented using <b>abstract classes</b> and <b>interfaces</b> .	It can be implemented by using the <b>access modifiers</b> (private, public, protected).
⑤ It is the process of <b>gaining</b> information.	It is the process of <b>containing</b> the information.
⑥ In abstraction, we use <b>abstract classes</b> and <b>interfaces</b> to hide the code complexities.	We use the <b>getters</b> and <b>setters</b> methods to hide the data.
⑦ The objects are <b>encapsulated</b> that helps to perform abstraction.	The object need not to <b>abstract</b> that result in encapsulation.

→ Data hiding - data security + hiding complexity (private)  
→ Encapsulation - hides the complexity of the system only (hiding public/private)  
↓  
process in data hiding

Example of encapsulation → Encryption

Example of Abstraction → using the string method  
(we need not know how it stores objects data inside the class)

# Object Oriented Programming IV

## Access Control, In-built packages, Object Class

1 Access Control:

2  
3 How a member can be accessed is determined by the access modifier attached to its declaration.  
4 Usually, you will want to restrict access to the data members of a class—allowing access only through  
methods.

5 Also, there will be times when you will want to define methods that are private to a class.

6  
7 Java's access modifiers are public, private, and protected. Java also defines a default access level.  
8 protected applies only when inheritance is involved.

9  
10 When no access modifier is used, then by default the member of a class is public within its own  
package,

11 but cannot be accessed outside of its package.

	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	World (diff pkg & not subclass)
public	+	+	+	+	+
protected	+	+	+	+	
no modifier	+	+	+		
private	+				

for sensitive data  
use by public getters  
setters

24 + : accessible  
25 blank : not accessible

26  
27 package packageOne;  
28 public class Base  
29 {  
30 protected void display(){  
31 System.out.println("in Base");  
32 }  
33 }  
34  
35 package packageTwo;  
36 public class Derived extends packageOne.Base{  
37 public void show(){  
38 new Base().display(); // this is not working  
39 new Derived().display(); // is working  
40 display(); // is working  
41 }  
42 }  
43  
44 protected allows access from subclasses and from other classes in the same package.  
45 We can use child class to use protected member outside the package but only child class object can  
access it.  
46 That's why any Derived class instance can access the protected method in Base.  
47 The other line creates a Base instance (not a Derived instance!!).  
48 And access to protected methods of that instance is only allowed from objects of the same package.

```

50 display();
51 -> allowed, because the caller, an instance of Derived has access to protected members and fields of
52 its subclasses,
53 even if they're in different packages
54
55 new Derived().display();
56 -> allowed, because you call the method on an instance of Derived and that instance has access to the
57 protected methods
58 of its subclasses
59
60 new Base().display();
61 -> not allowed because the caller's (the this instance) class is not defined in the same package like
62 the Base class,
63 so this can't access the protected method. And it doesn't matter – as we see – that the current
64 subclasses a class from
65 that package. That backdoor is closed ;)
66
67
68 class C
69     protected member;
70
71 // in a different package
72
73 class S extends C
74
75     obj.member; // only allowed if type of obj is S or subclass of S
76
77 The motivation is probably as following. If obj is an S, class S has sufficient knowledge of its
78 internals,
79 it has the right to manipulate its members, and it can do this safely.
80 If obj is not an S, it's probably another subclass S2 of C, which S has no idea of.
81 S2 may have not even been born when S is written. For S to manipulate S2's protected internals is quite
82 dangerous.
83 If this is allowed, from S2's point of view, it doesn't know who will tamper with its protected
84 internals and how,
85 this makes S2 job very hard to reason about its own state.
86
87 Now if obj is D, and D extends S, is it dangerous for S to access obj.member? Not really.
88 How S uses member is a shared knowledge of S and all its subclasses, including D. S as the superclass
89 has the right to
90 define behaviours, and D as the subclass has the obligation to accept and conform.
91
92 For easier understanding, the rule should really be simplified to require obj's (static) type to be
93 exactly S.
94 After all, it's very unusual and inappropriate for subclass D to appear in S. And even if it happens,
95 that the static type of obj is D, our simplified rule can deal with it easily by upcasting:
96 ((S)obj).member

```

# In built packages in Java →

- Lang - contains language specific stuff, ex- add, subtract, multiply and specific Java, no need to import this.
- io → File reading, optimise the input output, BufferedReader
- util → contains data structures like ArrayList and collections framework.
- applet
- awt - gui
- net - networking stuff.

Object class → Inside the lang class

- Top most class in inheritance, every class by default extends this class.  
even if multiple inheritance not allowed, but this is an exception, happens internally.
- Object class methods -
  - `toString`, `finalize`, `hashCode`  
`equals`,  
    ↳ random value ↗ algorithmic value
- `instanceof`  
    ↳ tells whether something is a subclass of something
- `object.getClass();` → get data from a class  
    ↳ can't override this

# Object Oriented Programming V

## Abstract classes , Interfaces , Annotations

1 Sometimes you will want to create a superclass that only defines a generalized form that will be shared  
2 by all of its  
3 subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of  
the methods that  
4 the subclasses must implement.  
5 You may have methods that must be overridden by the subclass in order for the subclass to have any  
meaning.  
6 In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods.  
Java's solution  
7 to this problem is the abstract method.  
8 You can require that certain methods be overridden by subclasses by specifying the abstract type  
modifier.  
9     abstract type name(parameter-list);  
10  
11 These methods are sometimes referred to as subclass's responsibility because they have no  
implementation specified in  
12 the superclass.  
13 Thus, a subclass must override them—it cannot simply use the version defined in the superclass.  
14  
15 Any class that contains one or more abstract methods must also be declared abstract.

```
package com.kunal.abstractDemo;

public class Main {
    public static void main(String[] args) {
        Son son = new Son();
        son.career();

        Daughter daughter = new Daughter();
        daughter.career();
    }
}
```

main.java

```
package com.kunal.abstractDemo;

public abstract class Parent {
    abstract void career(String name);
    abstract void partner(String name, int age);
}
```

parent.java

```
package com.kunal.abstractDemo;

public class Daughter extends Parent{
    2 related problems
    @Override
    void career() {
        System.out.println("I am going to be a coder");
    }

    @Override
    void partner() {
        System.out.println("I love Iron Man");
    }
}
```

daughter.java

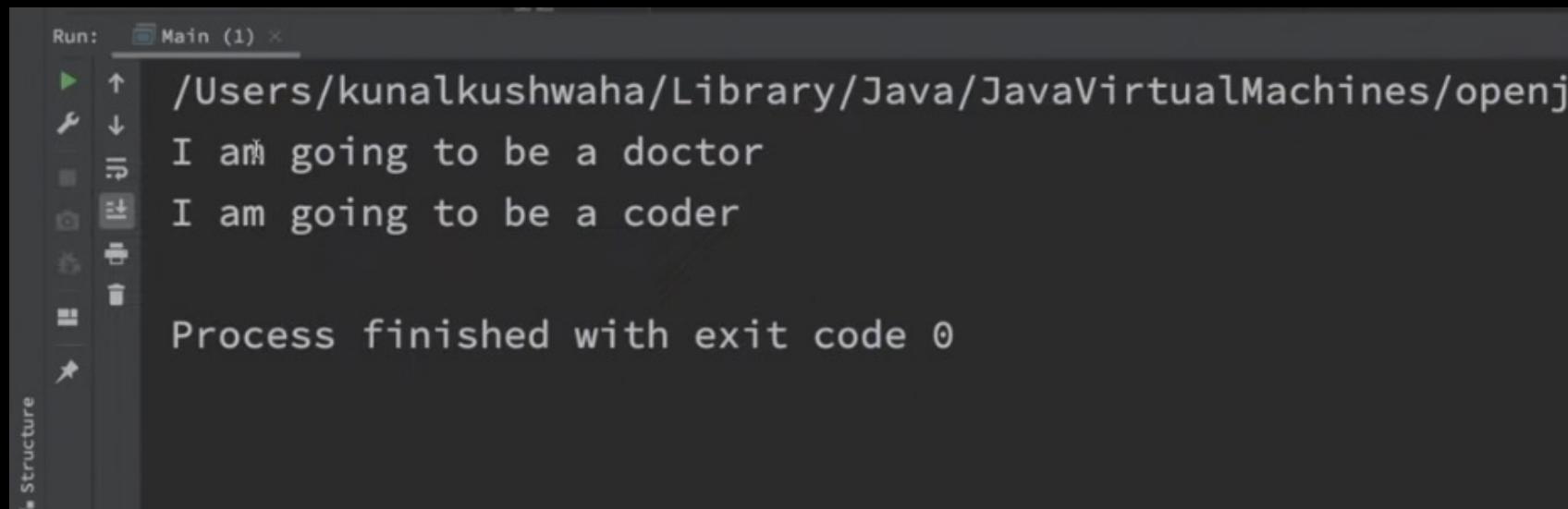
```
package com.kunal.abstractDemo;

public class Son extends Parent{
    2 related problems
    @Override
    void career() {
        System.out.println("I am going to be a doctor");
    }

    @Override
    void partner() {
        System.out.println("I love Pepper Potts");
    }
}
```

Son.java

## Output →



```
Run: Main (1) ×  
/Users/kunalkushwaha/Library/Java/JavaVirtualMachines/openj  
I am going to be a doctor  
I am going to be a coder  
  
Process finished with exit code 0
```

- We cannot create objects of an abstract class.
  - ↳ we override methods with it, coz methods inside won't have any body
- We also cannot create constructors of abstract class
  - + no static abstract methods
- We can create static methods in abstract classes

```
16 # There can be no objects of an abstract class.  
17 # You cannot declare abstract constructors, or abstract static methods.  
18 # You can declare static methods in abstract class.  
19 Because there can be no objects for abstract class. If they had allowed to call abstract static  
methods,  
20 it would mean we are calling an empty method (abstract) through classname because it is static.  
21 Any subclass of an abstract class must either implement all of the abstract methods in the superclass,  
22 or be declared abstract itself.  
23 Abstract classes can include as much implementation as they see fit i.e. there can be concrete  
methods(methods with body)  
24 in abstract class.  
  
26 Although abstract classes cannot be used to instantiate objects, they can be used to create object  
references,  
27 because Java's approach to run-time polymorphism is implemented through the use of superclass  
references.  
28  
29 A public constructor on an abstract class doesn't make any sense because you can't instantiate an  
abstract class directly  
30 (can only instantiate through a derived type that itself is not marked as abstract)
```

We can have constructor like this →  
using 'super keyword'  
→

Yes, an abstract class can have a constructor. Consider this:

```
abstract class Product {  
    int multiplyBy;  
    public Product( int multiplyBy ) {  
        this.multiplyBy = multiplyBy;  
    }  
  
    public int multiply(int val) {  
        return multiplyBy * val;  
    }  
}  
  
class TimesTwo extends Product {  
    public TimesTwo() {  
        super(2);  
    }  
}  
  
class TimesWhat extends Product {  
    public TimesWhat(int what) {  
        super(what);  
    }  
}
```

The superclass `Product` is abstract and has a constructor. The concrete class `TimesTwo` has a constructor that just hardcodes the value 2. The concrete class `TimesWhat` has a constructor that allows the caller to specify the value.

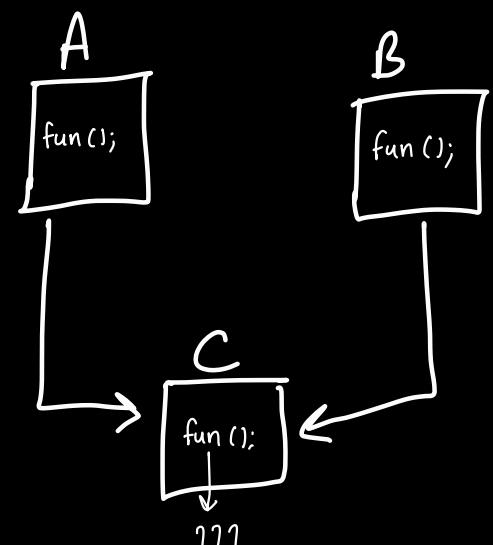
Abstract constructors will frequently be used to enforce class constraints or invariants such as the minimum fields required to setup the class.

NOTE: As there is no default (or no-arg) constructor in the parent abstract class, the constructor used in subclass must explicitly call the parent constructor.

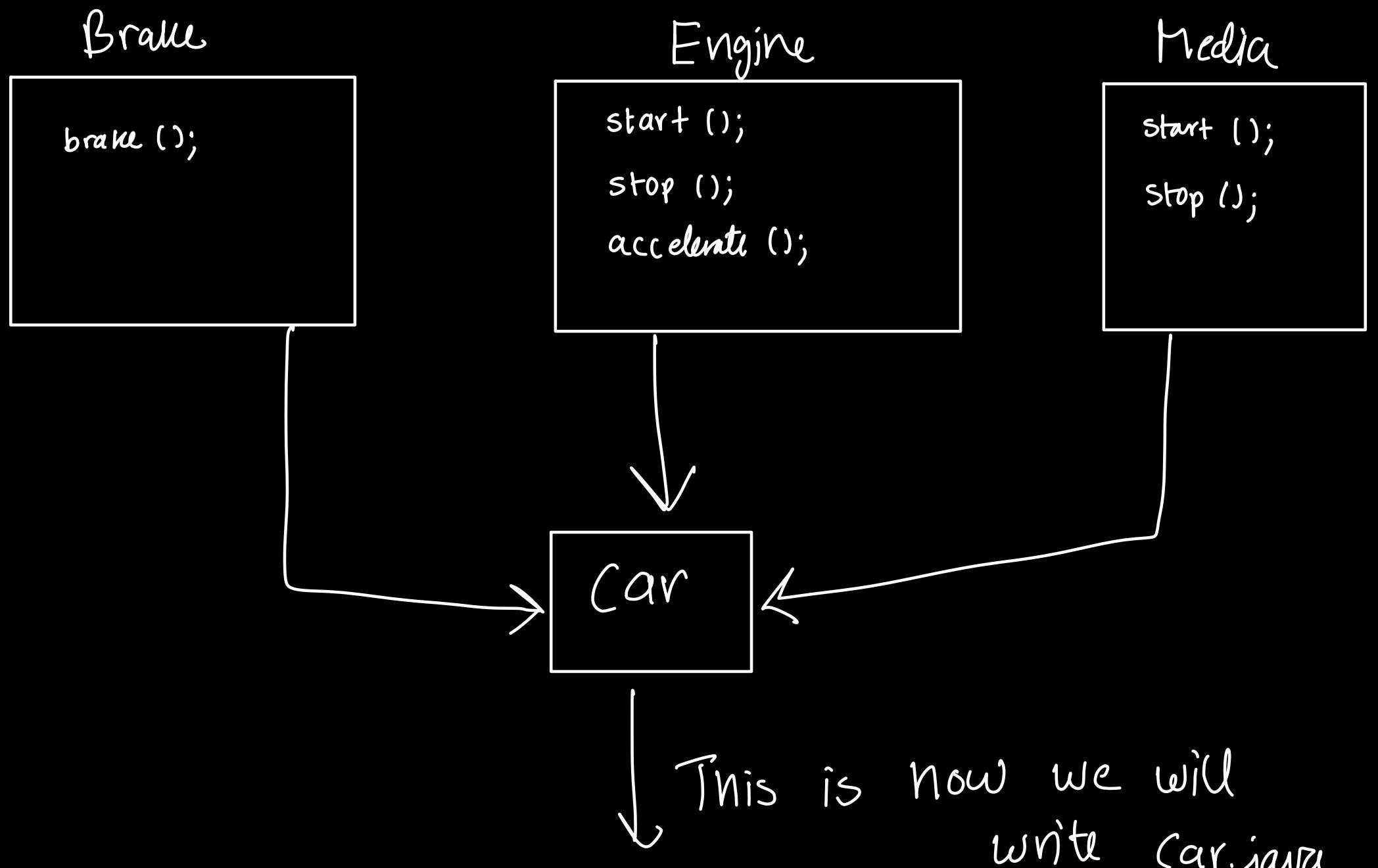
Interfaces — contains abstract classes

↳ Solves multiple inheritance

→ variables in this are  
static and final!



Example →



```
package com.kunal.interfaces;

public class Car implements Engine, Brake{
    @Override
    public void brake() {
        System.out.println("I brake like a normal Car");
    }

    @Override
    public void start() {
        System.out.println("I start like a normal Car");
    }

    @Override
    public void stop() {
        System.out.println("I stop like a normal Car");
    }

    @Override
    public void acc() {
        System.out.println("I accelerate like a normal Car");
    }
}
```

This is how we  
will write  
engine.java

```
package com.kunal.interfaces;

public interface Engine {
    static final int PRICE = 78000;
    void start();
    void stop();
    void acc();
}
```

1 Multiple inheritance is not available in java.  
2 (Same functions in 2 classes it will skip that hence no multiple inheritance)

4 Instead we have java interfaces. they have abstract functions (no body of functions)

6 Interface is like class but not completely. it is like an abstract class.  
7 By default functions are public and abstract in interface.  
8 variables are final and static by default in interface.

10 Interfaces specify only what the class is doing, not how it is doing it.  
11 The problem with MULTIPLE INHERITANCE is that two classes may define different ways of doing the same thing,  
12 and the subclass can't choose which one to pick.

14 Key difference between a class and an interface: a class can maintain state information  
15 (especially through the use of instance variables), but an interface cannot.

17 Using interface, you can specify a set of methods that can be implemented by one or more classes.  
18 Although they are similar to abstract classes, interfaces have an additional capability:  
19 A class can implement more than one interface. By contrast, a class can only inherit a single  
superclass  
20 (abstract or otherwise).

22 Using the keyword interface, you can fully abstract a class' interface from its implementation.  
23 That is, using interface, you can specify what a class must do, but not how it does it.

25 Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general  
rule,  
26 their methods are declared without any body.

28 By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple  
methods"  
aspect of polymorphism.

--  
31 NOTE: Interfaces are designed to support dynamic method resolution at run time.  
32 Normally, in order for a method to be called from one class to another, both classes need to be  
present at compile time  
33 so the Java compiler can check to ensure that the method signatures are compatible. This requirement  
by itself makes for  
34 a static and nonextensible classing environment. Inevitably in a system like this, functionality gets  
pushed up higher  
35 and higher in the class hierarchy so that the mechanisms will be available to more and more  
subclasses. Interfaces are  
36 designed to avoid this problem. They disconnect the definition of a method or set of methods from the  
inheritance  
37 hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that  
are unrelated  
38 in terms of the class hierarchy to implement the same interface. This is where the real power of  
interfaces is realized.

40 Beginning with JDK 8, it is possible to add a default implementation to an interface method.  
41 Thus, it is now possible for interface to specify some behavior. However, default methods constitute  
what is, in essence,  
42 a special-use feature, and the original intent behind interface still remains.

44 Variables can be declared inside of interface declarations.  
45 NOTE: They are implicitly final and static, meaning they cannot be changed by the implementing class.  
46 They must also be initialized. All methods and variables are implicitly public.

48 NOTE: The methods that implement an interface must be declared public. Also, the type signature of the  
implementing  
49 method must match exactly the type signature specified in the interface definition.

51 It is both permissible and common for classes that implement interfaces to define additional members  
of their own.

→ type of variable as type of interface allowed

Engine car = new Car();

↓

What things you access depend on this

which versions to access depend on this  
if both engine and car have eg- acc  
∴ the acc inside car will be used instead of engine acc var.

now Car class has

int a = 30;

in function Main,

Reason

we cannot access car.a;

→ watch OOP 5, 43:00 - 54:00

53 NOTE:  
54 You can declare variables as object references that use an interface rather than a class type.  
55 This process is similar to using a superclass reference to access a subclass object.  
56 Any instance of any class that implements the declared interface can be referred to by such a variable.  
57 When you call a method through one of these references, the correct version will be called based on the actual instance  
58 of the interface being referred to. Called at run time by the type of object it refers to.  
59 The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which  
60 calls methods on them.  
61 The calling code can dispatch through an interface without having to know anything about the "callee."  
62  
63 CAUTION: Because dynamic lookup of a method at run time incurs a significant overhead when compared with the normal  
64 method invocation in Java, you should be careful not to use interfaces casually in performance-critical code.  
65

# Now → Example

## ① CDPlayer.java

```

1 package com.kunal.interfaces;
2
3 public class CDPlayer implements Media{
4
5     @Override
6     public void start() {
7         System.out.println("Music start");
8     }
9
10    @Override
11    public void stop() {
12        System.out.println("Music stop");
13    }
14}

```

## ② Electric Engine . Java

```

1 package com.kunal.interfaces;
2
3 public class ElectricEngine implements Engine{
4
5     @Override
6     public void start() {
7         System.out.println("Electric engine start");
8     }
9
10    @Override
11    public void stop() {
12        System.out.println("Electric engine stop");
13    }
14
15    @Override
16    public void acc() {
17        System.out.println("Electric engine accelerate");
18    }
19}

```

## ③ Power Engine . Java

```

1 package com.kunal.interfaces;
2
3 public class PowerEngine implements Engine{
4
5     @Override
6     public void start() {
7         System.out.println("Power engine start");
8     }
9
10    @Override
11    public void stop() {
12        System.out.println("Power engine stop");
13    }
14
15    @Override
16    public void acc() {
17        System.out.println("Power engine accelerate");
18    }
19}

```

## ④ NiceCar.java

```

1 package com.kunal.interfaces;
2
3 public class NiceCar {
4     private Engine engine;
5     private Media player = new CDPlayer();
6
7     public NiceCar() {
8         engine = new PowerEngine();
9     }
10
11    public NiceCar(Engine engine) {
12        this.engine = engine;
13    }
14
15    public void start() {
16        engine.start();
17    }
18
19    public void stop() {
20        engine.stop();
21    }
22
23    public void startMusic() {
24        player.start();
25    }
26
27    public void stopMusic() {
28        player.stop();
29    }
30
31    public void upgradeEngine() {
32        this.engine = new ElectricEngine();
33    }
34}

```

## ⑤ Main.java

```

13
14     NiceCar car = new NiceCar();
15
16     car.start(); → refers to power engine
17     car.startMusic();
18     car.upgradeEngine(); → now diff constructor used
19     car.start();
20 }
21

```

*(diff parameter passed)*

Output →

Power engine start  
Music start  
Electric engine start

JOIN THE WAR

# Extending Interfaces →

```
97 Interfaces Can Be Extended:  
98 One interface can inherit another by use of the keyword extends. The syntax is the same as for  
99 inheriting classes.  
100 Any class that implements an interface must implement all methods required by that interface,  
101 including any that are  
inherited from other interfaces.
```

Annotations en - @Override, it is interface public  
internally

## Default

Methods + imp notes

```
102  
103 Default Interface Methods (aka extension method) :  
104 A primary motivation for the default method was to provide a means by which interfaces could be  
expanded without breaking existing code.  
105 i.e. suppose you add another method without body in an interface. Then you will have to provide the  
body of that method  
106 in all the classes that implement that interface.  
107 Ex:  
108     default String getString() {  
109         return "Default String";  
110     }  
111  
112 For example, you might have a class that implements two interfaces.  
113 If each of these interfaces provides default methods, then some behavior is inherited from both.  
114 # In all cases, a class implementation takes priority over an interface default implementation.  
115 # In cases in which a class implements two interfaces that both have the same default method, but the  
class does not  
116 override that method, then an error will result.  
117 # In cases in which one interface inherits another, with both defining a common default method, the  
inheriting  
118 interface's version of the method takes precedence.  
119 IMP  
120 NOTE: static interface methods are not inherited by either an implementing class or a subinterface.  
121 i.e. static interface methods should have a body! They cannot be abstract.  
122  
123 NOTE : when overriding methods, the access modifier should be same or better i.e. if in Parent Class  
it was protected, then then overridden should be either protected or public.  
124
```

### A. Java

```
1 package com.kunal.interfaces.extendDemo2;  
2  
3 public interface A {  
4     // static interface methods should always have a body  
5     // call via the interface name  
6     static void greeting() {  
7         System.out.println("Hey I am static method");  
8     }  
9  
10    default void fun() {  
11        System.out.println("I am in A");  
12    }  
13}
```

### B. Java

```
1 package com.kunal.interfaces.extendDemo2;  
2  
3 public interface B{  
4     void greet();  
5  
6     // default void fun() {  
7     //     System.out.println("I am in A");  
8     // }  
9  
10    // void fun();  
11}
```

## Main.java

```
1 package com.kunal.interfaces.extendDemo2;
2
3 public class Main implements A, B {
4     @Override
5     public void greet() {
6
7     }
8
9     public static void main(String[] args) {
10         Main obj = new Main();
11         A.greeting();
12     }
13 }
```

## Nested Interfaces → A.java

```
1 package com.kunal.interfaces.nested;
2
3 public class A {
4     // nested interface
5     public interface NestedInterface {
6         boolean isOdd(int num);
7     }
8
9
10 class B implements A.NestedInterface {
11     @Override
12     public boolean isOdd(int num) {
13         return (num & 1) == 1;
14     }
15 }
```

```
1 package com.kunal.interfaces.nested;
2
3 public class Main {
4     public static void main(String[] args) {
5         B obj = new B();
6         System.out.println(obj.isOdd(6));
7     }
8 }
```

main.java

Notes ↴

67 Nested Interfaces:

68

69 An interface can be declared a member of a class or another interface. Such an interface  
70 is called a member interface or a nested interface. A nested interface can be declared as public,  
private, or protected.

71 This differs from a top-level interface, which must either be declared as public or use the default  
access level.

72

73 // This class contains a member interface.

74 class A {

75 // this is a nested interface

76 public interface NestedIF {

77 boolean isNotNegative(int x);

78 }

79 }

80 // B implements the nested interface.

81 class B implements A.NestedIF {

82 public boolean isNotNegative(int x) {

83 return x < 0 ? false : true;

84 }

85 }

86 class NestedIFDemo {

87 public static void main(String args[]) {

88 // use a nested interface reference

89 A.NestedIF nif = new B();

90 if(nif.isNotNegative(10))

91 System.out.println("10 is not negative");

92 if(nif.isNotNegative(-12))

93 System.out.println("this won't be displayed");

94 }

95 }

# Abstract Classes

V/S

# Interface

```
33
34 Abstract class vs Interface:
35
36 Type of methods:
37 Interface can have only abstract methods.
38 Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static
methods also.
39
40 Final Variables:
41 Variables declared in a Java interface are by default final.
42 An abstract class may contain non-final variables.
43
44 Type of variables:
45 Abstract class can have final, non-final, static and non-static variables.
46 Interface has only static and final variables.
47
48 Implementation:
49 Abstract class can provide the implementation of interface.
50 Interface can't provide the implementation of abstract class.
51
52 Inheritance vs Abstraction:
53 A Java interface can be implemented using keyword "implements"
54 and abstract class can be extended using keyword "extends".
55
56 Multiple implementation:
57 An interface can extend another Java interface only,
58 an abstract class can extend another Java class and implement multiple Java interfaces.
59
60 Accessibility of Data Members:
61 Members of a Java interface are public by default.
62 A Java abstract class can have class members like private, protected, etc.
```

# Object Oriented Programming VI

Generics, Custom ArrayList, Lambda Expressions  
Exception handling, object cloning.

```
1 package com.kunal.generics;
2
3 import java.util.ArrayList;
4 import java.util.Arrays;
5
6 public class CustomArrayList {
7
8     private int[] data;
9     private static int DEFAULT_SIZE = 10;
10    private int size = 0; // also working as index value
11
12    public CustomArrayList() {
13        this.data = new int[DEFAULT_SIZE];
14    }
15
16    public void add(int num) {
17        if (isFull()) {
18            resize();
19        }
20        data[size++] = num;
21    }
22
23    private void resize() {
24        int[] temp = new int[data.length * 2];
25
26        // copy the current items in the new array
27        for (int i = 0; i < data.length; i++) {
28            temp[i] = data[i];
29        }
30        data = temp;
31    }
32
33    private boolean isFull() {
34        return size == data.length;
35    }
36
37    public int remove() {
38        int removed = data[--size];
39        return removed;
40    }
41
42    public int get(int index) {
43        return data[index];
44    }
45
46    public int size() {
47        return size;
48    }
49
50    public void set(int index, int value) {
51        data[index] = value;
52    }
```

here we have made  
a custom ArrayList →

→ creates the List  
with default size

→ checks if List is  
full and then doubles it  
using resize

→ This doubles the data  
of the arrayList

→ general imp  
features of  
ArrayList

```

54     @Override
55     public String toString() {
56         return "CustomArrayList{" +
57             "data=" + Arrays.toString(data) +
58             ", size=" + size +
59             '}';
60     }
61
62     public static void main(String[] args) {
63         //     ArrayList list = new ArrayList();
64         CustomArrayList list = new CustomArrayList();
65         //     list.add(3);
66         //     list.add(5);
67         //     list.add(9);
68
69         for (int i = 0; i < 14; i++) {
70             list.add(2 * i);
71         }
72
73         System.out.println(list);
74
75         ArrayList<Integer> list2 = new ArrayList<>();
76         //     list2.add("dfghj");
77     }
78 }
```

→ printing the  
ArrayList

Problem

→ only integer  
ArrayList can be  
created

Called as Generic

Solved  
using

## Generics →

ArrayList < Integer > list = new ArrayList <>();  
 ↓  
 generic , can't be a primitive data type  
 provides type safety .

generally we  
 add < T > in the  
 class name like this  
 < T > replaced  
 by later < Integer >

```

8  public class CustomGenArrayList<T> {
9
10    private Object[] data;
11    private static int DEFAULT_SIZE = 10;
12    private int size = 0; // also working as index value
13
14    public CustomGenArrayList() {
15        data = new Object[DEFAULT_SIZE];
16    }
17 }
```

```

18 public void add(T num) {
19     if (isFull()) {
20         resize();
21     }
22     data[size++] = num;
23 }
24
25 private void resize() {
26     Object[] temp = new Object[data.length * 2];
27
28     // copy the current items in the new array
29     for (int i = 0; i < data.length; i++) {
30         temp[i] = data[i];
31     }
32     data = temp;
33 }
34
35 private boolean isFull() {
36     return size == data.length;
37 }
38
39 public T remove() {
40     T removed = (T)(data[--size]);
41     return removed;
42 }
43
44 public T get(int index) {
45     return (T)data[index];
46 }
47

```

Array Created at Runtime  
if we put  $T[\text{Default\_Size}]$ ;

↓  
will give error

Because at compile time it  
doesn't know what  $T$  is  
type is checked at  
compile time only

↓  
Type safety would have  
been a problem

Java uses type erasure. → replacing  $T$  with  $\text{Integer}$

Type erasure can be explained as the process of enforcing type constraints only at compile time and discarding the element type information at runtime.

For example:

```

public static <E> boolean containsElement(E [] elements, E element){
    for (E e : elements){
        if(e.equals(element)){
            return true;
        }
    }
    return false;
}

```

The compiler replaces the unbound type  $E$  with an actual type of  $\text{Object}$ .

```

public static boolean containsElement(Object [] elements, Object element){
    for (Object e : elements){
        if(e.equals(element)){
            return true;
        }
    }
    return false;
}

```

Therefore the compiler ensures type safety of our code and prevents runtime errors.

```
9 // here T should either be Number or its subclasses  
10 public class WildcardExample<T extends Number> {  
11 }
```

wildcard example ↑ only number class allowed  
like String not allowed.

```
19  
20     public void getList(List<? extends Number> list) {  
21         // do something  
22     }
```

only <Number>  
↓  
can now allow subclasses of  
number like float, etc.  
↳ only Int, Double ...

## Comparing Objects

main.java

```
1 package com.kunal.generics.comparing;  
2  
3 import java.util.Arrays;  
4 import java.util.Comparator;  
5  
6 public class Main {  
7     public static void main(String[] args) {  
8         Student kunal = new Student(12, 89.76f);  
9         Student rahul = new Student(5, 99.52f);  
10        Student arpit = new Student(2, 95.52f);  
11        Student karan = new Student(13, 77.52f);  
12        Student sachin = new Student(9, 96.52f);  
13  
14        Student[] list = {kunal, rahul, arpit, karan, sachin};  
15  
16        System.out.println(Arrays.toString(list));  
17        // Arrays.sort(list, new Comparator<Student>() {  
18        //     @Override  
19        //     public int compare(Student o1, Student o2) {  
20        //         return -(int)(o1.marks - o2.marks);  
21        //     }  
22        // });  
23        //     ↳ makes ascending descending  
24        Arrays.sort(list, (o1, o2) -> -(int)(o1.marks - o2.marks));  
25  
26        System.out.println(Arrays.toString(list));  
27  
28        //     if (kunal.compareTo(rahul) < 0) {  
29        //         System.out.println(kunal.compareTo(rahul));  
30        //         System.out.println("Rahul has more marks");  
31        //     }  
32  
33    }
```

↑ same as this  
but uses Lambda  
expression

## Student.java

```
1 package com.kunal.generics.comparing;  
2  
3 public class Student implements Comparable<Student>{  
4     int rollno;  
5     float marks;  
6  
7     public Student(int rollno, float marks) {  
8         this.rollno = rollno;  
9         this.marks = marks;  
10    }  
11  
12    @Override  
13    public String toString() {  
14        return marks + "";  
15    }  
16  
17    @Override  
18    public int compareTo(Student o) {  
19        System.out.println("in compareto method");  
20        int diff = (int)(this.marks - o.marks);  
21  
22        // if diff == 0: means both are equal  
23        // if diff < 0: means o is bigger else o is smaller  
24  
25        return diff;  
26    }  
27}
```

*↑ comparing method*

*does the comparison*

## Lambda Functions →

```
1 package com.kunal.generics;  
2  
3 import java.util.ArrayList;  
4 import java.util.function.Consumer;  
5  
6 public class LambdaFunctions {  
7     public static void main(String[] args) {  
8         ArrayList<Integer> arr = new ArrayList<>();  
9         for (int i = 0; i < 5; i++) {  
10             arr.add(i + 1);  
11         }  
12         arr.forEach((item) -> System.out.println(item * 2));  
13  
14         Consumer<Integer> fun = (item) -> System.out.println(item * 2);  
15         arr.forEach(fun);  
16  
17         Operation sum = (a, b) -> a + b;  
18         Operation prod = (a, b) -> a * b;  
19         Operation sub = (a, b) -> a - b;
```

*runs a for loop*

*prints every item after multiplying it by 2*

*←*

```

21 LambdaFunctions myCalculator = new LambdaFunctions();
22 System.out.println(myCalculator.operate(5, 3, sum));
23 System.out.println(myCalculator.operate(5, 3, prod));
24 System.out.println(myCalculator.operate(5, 3, sub));
25 }
26     ↘ comes here ①
27     private int operate(int a, int b, Operation op) {
28         return op.operation(a, b);
29     }
30 }
31 interface Operation {
32     int operation(int a, int b);
33 }
34

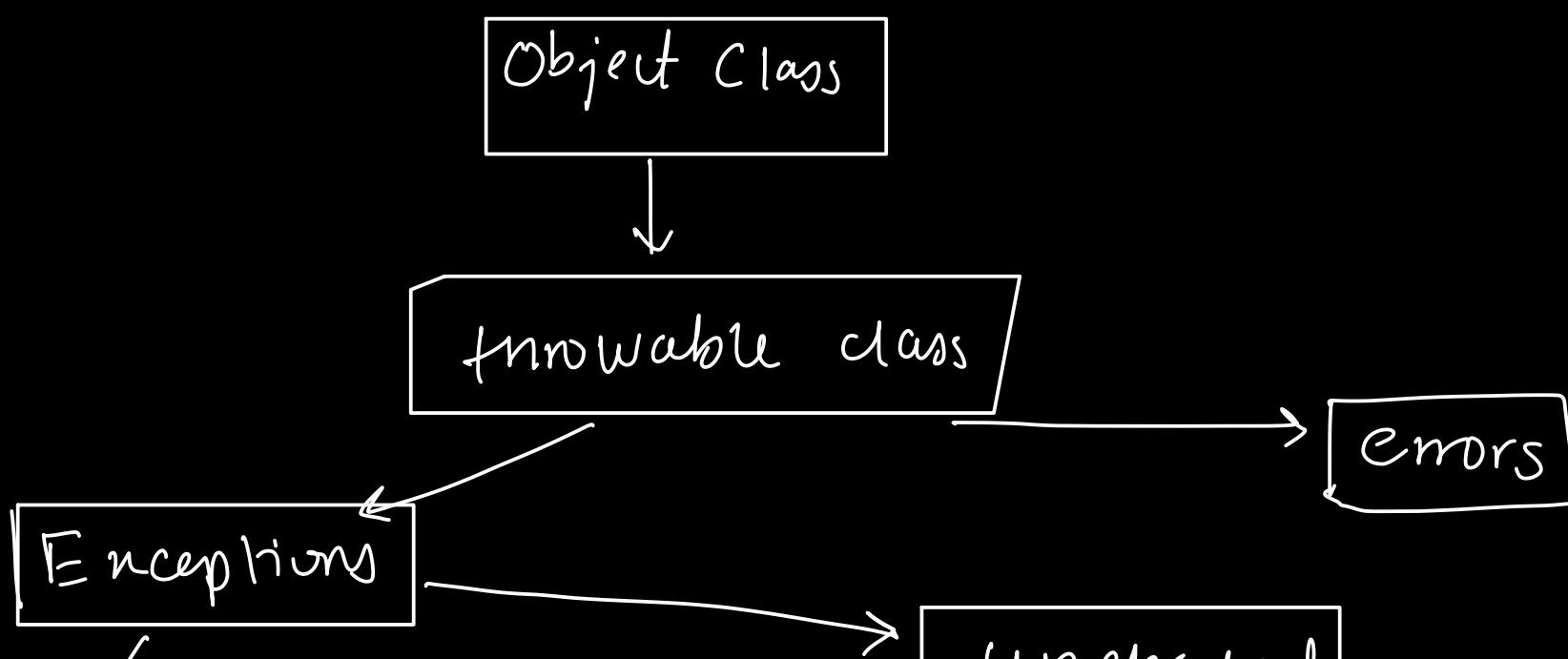
```

Then goes to its body here.

↙ then here ②

Exception handling - errors basically can't be corrected  
 ↓  
 ∵ e.g. stack overflow etc.

Stops the flow, example dividing by zero.



happens at compile time, like filenot found.

happens at runtime  
 e.g. divide by zero arithmetic exception.

```

1 package com.kunal.exceptionHandling;
2
3 public class Main {
4     public static void main(String[] args) {
5         int a = 5;
6         int b = 0;
7         try {
8             // divide(a, b);
9             // mimicing
10            String name = "Kunal";
11            if (name.equals("Kunal")) {
12                throw new MyException("name is Kunal");
13            }
14            } catch (MyException e) {
15                System.out.println(e.getMessage());
16            } catch (ArithmaticException e) {
17                System.out.println(e.getMessage());
18            } catch (Exception e) {
19                System.out.println("normal exception");
20            } finally {
21                System.out.println("this will always execute");
22            }
23        }
24
25
26        static int divide(int a, int b) throws ArithmaticException{
27            if (b == 0) {
28                throw new ArithmaticException("please do no divide by zero");
29            }
30            return a / b;
31        }
32    }
33

```

More strict rule above (more specific)  
comes below

*myexception.java*

custom exception class  
calls constructor.

pretty prints the error message  
(catches it and prints it)

finally always executes  
only one finally allowed

means we are throwing an exception.  
Knows - used to declare exceptions (may throw)

→ custom exception handling

Object Cloning → enact copies of an object much faster.

main.java

```

1 package com.kunal.cloning;
2
3 import java.util.Arrays;
4
5 public class Main {
6     public static void main(String[] args) throws CloneNotSupportedException {
7         Human kunal = new Human(34, "Kunal Kushwaha");
8         // Human twin = new Human(kunal); → simply copies the object, Slow
9
10        Human twin = (Human)kunal.clone();
11        System.out.println(twin.age + " " + twin.name);
12        System.out.println(Arrays.toString(twin.arr));
13
14        twin.arr[0] = 100;
15
16        System.out.println(Arrays.toString(twin.arr));
17        System.out.println(Arrays.toString(kunal.arr));
18    }
19

```

anything inside function throws exception use this

## Human.java

```
1 package com.kunal.cloning;
2
3 public class Human implements Cloneable{
4     int age;
5     String name;
6     int[] arr;
7
8     public Human(int age, String name) {
9         this.age = age;
10        this.name = name;
11        this.arr = new int[]{3, 4, 5, 6, 9, 1};
12    }
13
14 //    @Override
15 //    public Object clone() throws CloneNotSupportedException{
16 //        // this is shallow copy
17 //        return super.clone();
18 //    }
19
20 @Override
21 public Object clone() throws CloneNotSupportedException{
22     // this is deep copy
23     Human twin = (Human)super.clone(); // this is actually shallow copy
24     // make a deep copy
25     twin.arr = new int[twin.arr.length];
26     for (int i = 0; i < twin.arr.length; i++) {
27         twin.arr[i] = this.arr[i];
28     }
29     return twin;
30 }
31
32 }
33 }
```

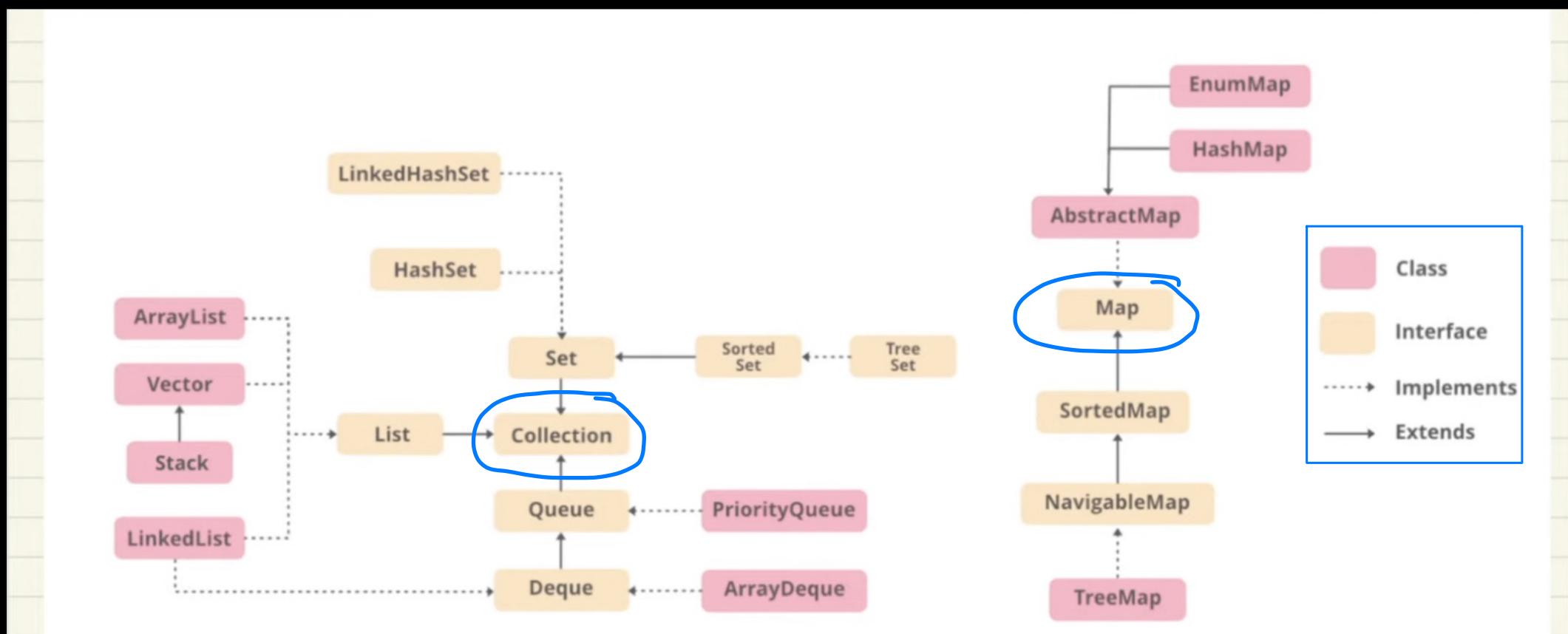
if cannot clone → copies stuff by java internally

Shallow Copy - it will copy the primitives  
but for arrays , it will point to  
the object array from which it copies

Deep Copy - creates a new object (array)

# Object Oriented Programming VII

## Collections framework, Vector Class, Enums in Java



functions like `get`, `remove`, made common to various data structures like linked list, stack queues etc. via abstraction is collections framework.

### Vectors

- Multiple threads can access the `ArrayList` at same time.
- single thread can access the `Vector` at a particular time while the other thread waits.

Code → internally it's diff, but same as `arraylist` on outside, `ArrayList` is fast

# Enums in Java →

```
1 An enumeration is a list of named constants.  
2 In Java, an enumeration defines a class type.  
3 By making enumerations into classes, the capabilities of the enumeration are greatly expanded.  
4  
5 An enumeration is created using the enum keyword.  
6 Enum declaration can be done outside a Class or inside a Class but not inside a Method  
7 We can declare main() method inside enum. Hence we can invoke enum directly from the Command Prompt.  
8  
9 /* internally above enum Color is converted to (Check Example.java)  
10 class Color  
11 {  
12     public static final Color Red = new Color();  
13     public static final Color Blue = new Color();  
14     public static final Color Green = new Color();  
15 }*/  
16  
17 Enum and Inheritance :  
18 -All enums implicitly extend java.lang.Enum class. As a class can only extend one parent in Java,  
19 so an enum cannot extend anything else.  
20 -An enum cannot be a superclass.  
21 -toString() method is overridden in java.lang.Enum class, which returns enum constant name.  
22 -enum can implement many interfaces.  
23  
24 Two enumeration constants can be compared for equality by using the == relational operator.  
25  
26 values(), ordinal() and valueOf() methods :  
27 These methods are present inside java.lang.Enum.  
28 -values() method can be used to return all values present inside enum.  
29 -Order is important in enums.By using ordinal() method, each enum constant index can be found,  
30 just like array index.  
31 -valueOf() method returns the enum constant of the specified string value, if exists.  
32  
33 enum and constructor :  
34 -enum can contain constructor and it is executed separately for each enum constant at the time  
35 of enum class loading.  
36 -We can't create enum objects explicitly and hence we can't invoke enum constructor directly.  
37 -And the constructor cannot be the public or protected it must have private or default modifiers.  
38 -Why? if we create public or protected, it will allow initializing more than one objects.  
39 -This is totally against enum concept.  
40  
41 enum and methods :  
42 enum can contain concrete methods only i.e. no any abstract method.  
43  
44 You can compare for equality an enumeration constant with any other object by using equals( ),  
45 which overrides the equals( ) method defined by Object.  
46 Although equals( ) can compare an enumeration constant to any other object, those two objects  
47 will be equal only if they both refer to the same constant,within the same enumeration.  
48 Simply having ordinal values in common will not cause equals( ) to return true if the two constants  
49 are from different enumerations. Remember, you can compare two enumeration references for equality by  
using ==.
```

Code - Sample → JOIN THE DARKSIDE

```
public class Basic {  
    enum Week {  
        Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday  
        // these are enum constants  
        // public, static and final  
        // since its final you can create child enums  
        // type is Week  
    }  
  
    public static void main(String[] args) {  
        Week week;  
        week = Week.Monday;  
  
        for(Week day : Week.values()) {  
            System.out.println(day);  
        }  
    }  
}
```

```
Week() {  
    System.out.println("Constructor called for " + this);  
}  
// this is not public or protected, only private or default  
// why? we don't want to create new objects  
// this is not the enum concept, that's why
```

↳ output of this constructor was  
all days printed