

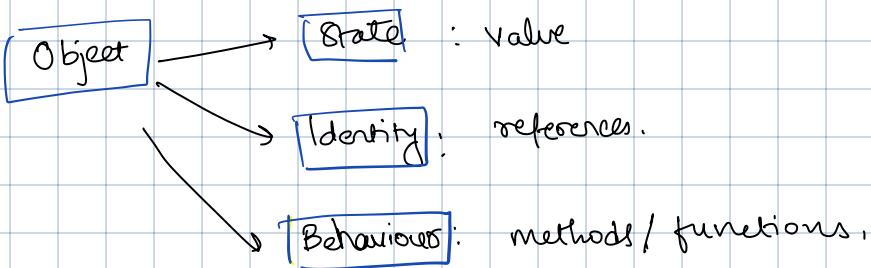
# **Object Oriented Programming (Java)**

- Shashidhar Nagaral

## Class.

Class is the template for the objects → logical construct.

Object is an instance of the class. → this is physical existence of a class.



How to access instance variables?

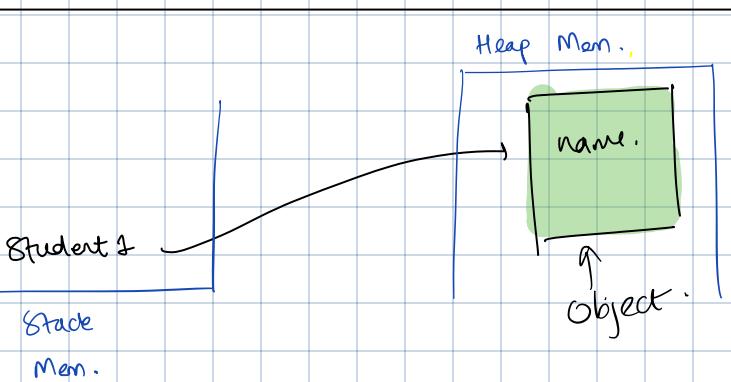
Student student1 = new Student();

↑ reference variable

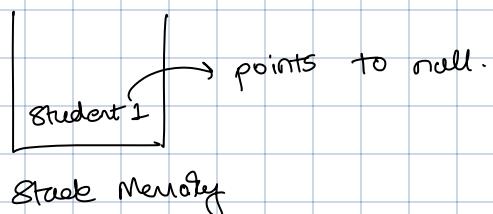
→ student1.name

```
class Student {  
    int rno;  
    String name;  
};
```

↑ instance variables



Student student1; // declaration.



NOTE: Student s1; // declaration, s1 will have NULL by default.

s1 = new Student(); // created object.  
↓  
dynamically allocates  
memory & returns the reference  
to it.

## Dynamic Memory Allocation:

Student student1 = new Student();

Compile time

Runtime

runtime means entire code  
is converted to bytecode if it  
actually running

NOTE: Student1 is not an object,  
it is a reference variable pointing  
to the object (in heap)

## Constructors

- it is special type of method.
- it is automatically called when the object is created, before the new operator completes.

Note: every class has a default constructor.

Box mybox1 = new Box();

→ new Box() is calling the Box() constructor

```
class Student {
```

```
    Student() {
```

```
        // default constructor
```

```
    }
```

```
}
```

"this" Keyword: this refers to instance of the class.

it is often used to differentiate between instance variables & parameters with same name.

Why don't we use "new" keyword for creating primitive data types?

In Java, the primitive data types like int, float, char, boolean are not implemented as objects.

Primitives



NOT  
objects



Not stored in Heap  
but stored in Stack

Then why does java have primitive data type?

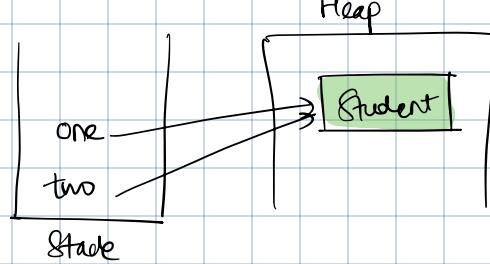
→ To increase the performance, memory efficiency ...

Memory Allocation of "new" Keyword:

Student one = new Student();

Student two = one;

i.e. one, two both are pointing to same instance of class present in the heap.



Wrapper Class:

int a = 20; // a is primitive

Integer num = 20; // num is an object.

NOTE:

Integer num = new Integer(10); X Not allowed from Java 9  
(Integer num = 10;) ✓ Since Java 9, there has been feature AUTO BOXING that automatically converts primitive types to their corresponding class & vice versa;

## Final

used to define constant, (immutable) → for optimisation.

NOTE: final + primitive → we cannot update the value

final + Non-primitive → In Objects, we cannot change the reference  
but we can change the value

## Packages:

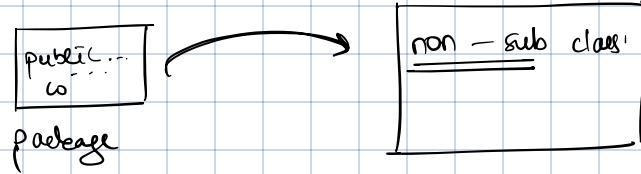
package are containers for classes

package is both a naming & a visibility control mechanism.

How does the Java run-time system know where to look for packages that you create?

- ① By default, the JRT system uses the current working directory as its starting point.
- ② we can specify a dir. path or paths by setting the CLASSPATH environmental variables
- ③ You can use the -classpath option with java and javac to specify the path to your classes.

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



## Static:

When a member is declared static, it can be accessed before any of its class are created.

most common example is main() method.

→ main() method 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 & not to the object.

NOTE: • A static method can access only static data.

• A static method can call other static method & cannot call a non-static method from it.

- A static method cannot refer to "this" or "super" keyword. in anyway

### Inner class:

Inner class act as a member to the outer class.

```

ex. class A { Outer class
    private int num;
    inner class B { String name;
        B(String name) {
            this.name = name;
        }
        num = 10; } we can access other member variables
    }
    pnum( ) { inside static inner class.
}

```

**X** `B b = new B("a");` // this will give error  
 bcz B is itself non static member of A and hence we can't access B without any instance of A.

**✓** `A a = new A();`  
`A.B b = a.new B("a");`

**X** `new a.B("a");` bcz we can't access B with class

**Static Inner class:** same as above code, but if the inner class is static then

- ① static tells that the inner class is not dependent on the instance of the outer class.
- ② we can't access other member variables inside static inner class.

```
class A { } // main()
```

```
static class B { String name;
    B(String name) {
        this.name = name;
    }
}
```

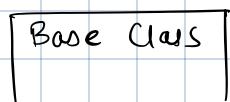
```
A.B b = new A.B();
```

A is just used to access the inner class

→ P8UM

## 4 Pillars of OOP's : Inheritance, Polymorphism, Encapsulation, Abstraction.

### Inheritance



```

class Parent {
    int length;
    int breadth;
}
  
```

```

class Child extends Parent {
    int price;
}
  
```

NOTE: • Java does not support multiple super class i.e. only one super class for any class can be specified.

### Private:

Subclass (child class) can access all of the members of its superclass, but cannot access members of the superclass that have been declared as private.

NOTE \*\* When a reference to a subclass object is assigned to a superclass reference variable; you will have access only to those parts of the object defined by superclass.

```

ex   class A {
        int a;
        int b;
    }
}
  
```

// main

super class  
reference type

```

class B extends A {
    int c;
    int d;
}
  
```

i.e. b-obj.c X

b-obj.a ✓

i.e. SUPERCLAS ref = new SUBCLASS();

Here ref can only access methods which are available in SUPERCLASS.

child reference → B a\_obj = new A(); X Error  
 type  
 this will show error bcz,

① Reference type is of child: this means parent object can access members of child class.

② Object is of Parent: but, instance is of parent class.  
 if parent does not access to its child class.

"SUPER": whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword "super".  
 it can be used for:

- ① to call superclass constructor
- ② access superclass members.

ex  
 child class → BoxWeight ( double w, double h ) {  
 super ( w, h ); ← constructor  
 super.w; ← superclass member.

NOTE: A superclass variable can be used to reference any object derived from that class.

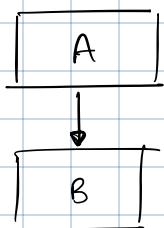
ex class Box {  
 double w;  
 double h;  
} we are passing  
object of type BoxWeight to the  
super of type Box

class BoxWeight extends Box {  
double weight;  
BoxWeight ( BoxWeight ob ) {  
super ( ob );  
weight = ob.weight;

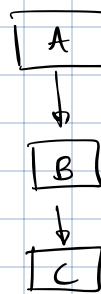
NOTE: If super() is not used in subclass's constructor, then default or parameters less constructor of each superclass will be executed.

## Inheritance types

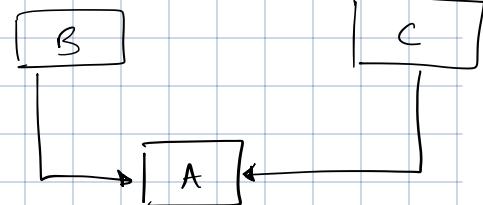
① Single Inheritance



② Multilevel Inheritance

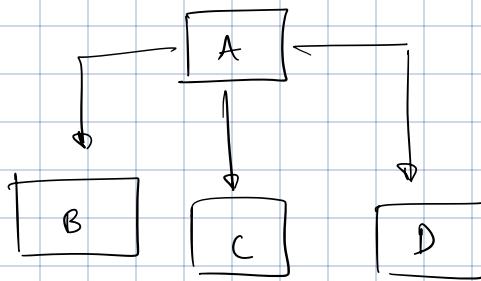


③ Multiple Inheritance X

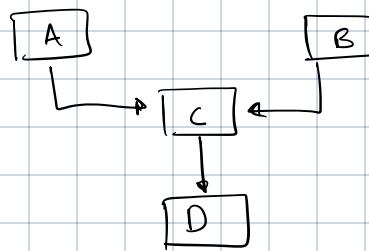


NOTE: Java does not support Multiple Inheritance

④ Hierarchical inheritance



X ⑤ Hybrid Inheritance: Single + multiple Inheritance



NOTE: We can achieve multiple inheritance through INTERFACE

## Polymorphism:

polymorphism allows objects of different classes to be treated as objects of a common superclass during runtime.

Type of Polymorphism:

① Compile time / Static Polymorphism (Method Overloading):

- method overloading in Java allows a class to have multiple methods with same name but different parameter list

Rules:

- ① Method names must be same.
- ② you can't overload a method just by changing its return type
- ③ The parameter lists must be different in terms of the number of parameters &/or their types.

## ② Runtime / dynamic polymorphism: (method overriding)

- Reference type is super class  
Obj type is subclass.

NOTE: Reference Variable type tells what all things are accessible

Type of object tells which one of those to be run.

How does overriding work?

Parent obj = new Child();  
Here, which method will be called depends on the type of object,  
this is known as Upcasting.

Class Shape of  
void area() {

}

Class Square extends Shape of

void area() { ← overridden  
the area()

Method of its  
superclass.

}

- Upcasting is the process of converting a reference variable of a subclass type to a reference variable of the superclass type.
- Upcasting is SAFE & implicit conversion because a subclass object inherently contains all the attributes & behaviors of its subclass.

Downcasting:

- Converting a reference variables of a superclass type to a reference variable of a subclass type.

- Not implicit, & requires an explicit cast.

- Downloading is needed when you want to access the specific methods & fields defined in the subclass.

ex. class Animal {  
 void makeSound();

// psum

}

Class Dog extends Animal {  
 void makeSound();

Animal animal = new Dog();  
animal.fetch() → since ref type is Animal, we can't have access to fetch method.

Soln: Downcasting

if (animal instanceof Dog) {

Dog dog = (Dog) animal;

dog.fetch();

}

How does java determines which members to be seen?

→ **Dynamic Method Dispatch**: also known as late binding, it allows the method invocation to be determined at runtime rather than compile-time.

NOTE: Private, final and static methods and variables uses static binding & bonded by compiler while overridden methods are bonded during runtime based upon type of runtime object.

### Encapsulation:

bundles the data (attribute) & methods (behavior) that operate on the data within single unit called a class.

- main idea behind encapsulation is to hide the internal details of an object; this is often referred to as data hiding.
- encapsulation is achieved using access modifiers & getter & setter methods.

Access Modifiers: control the visibility & accessibility of class members (fields & method).

Getter & setters: allow controlled access to the internal state of an object.

### Abstraction:

: it hides the internal implementation & shows only essential functionality to the user.

It can be achieved through **INTERFACE** & **ABSTRACT** classes.

ex. CAR: pedal, if we press it → car speed will increase  
BUT HOW? this is abstracted to us.

- Increase security & confidentiality

NOTE: Encapsulation : Implementation level issue.

Abstraction : Design level issue.

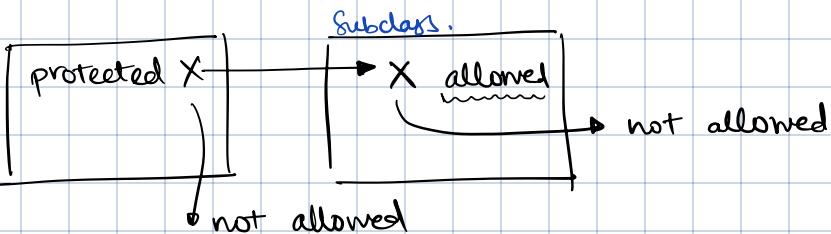
## Access Control :

for sensitive  $\leftarrow$  private: - not accessible outside the class.

data - to manipulate we use getter & setter method.



outside  $\leftarrow$  protected: - this makes the member to be accessed from package but only within subclass



use when we

want to  $\leftarrow$  default / package private: not accessible outside the package hide from outside package

we want to  $\leftarrow$  public: Accessible everywhere use everywhere

(Package Private)

| Access Modifier:        | Private | Default / no-access | Protected | Public |
|-------------------------|---------|---------------------|-----------|--------|
| Inside class            | ✓       | ✓                   | ✓         | ✓      |
| Same package class      | ✗       | ✓                   | ✓         | ✓      |
| Same package subclass   | ✗       | ✓                   | ✓         | ✓      |
| Other package subclass. | ✗       | ✗                   | ✓         | ✓      |
| Other package class     | ✗       | ✗                   | ✗         | ✓      |

**Abstract Class:** is a class that cannot be instantiated on its own.  
it's meant to be done by subclass or by other class.

- It serves as a blueprint for creating derived classes
- Abstract classes are often used to define a common base for a group of related classes  
abstract type name (parameter - list)
  - ↳ these methods are also referred as subclass's responsibility.

NOTE: • Any class that contains one or more abstract methods must also be declared abstract.  
• There can be no objects of an abstract class.

- You cannot declare abstract constructors, or abstract static methods.
- You can declare static method.
- Abstract classes can have concrete methods.

why?

bcz if it were allowed then it would mean that we are calling an empty method (abstract) through classname bcz it is static.

Abstract Classes can have public constructor but still they cannot be called then what is need to constructor inside abstract class?

- ① Abstract constructors will frequently be used to enforce class constraints or invariants such as minimum fields required to setup the class.

```
public abstract class Parent {  
    // can we make class final and abstract both?  
    // -> NO, since final doesn't allow extension of class, whereas abstract tells use to extend  
    // and override method, both contradict  
    abstract void run();  
  
    // abstract static void build(); // abstract and static is not possible because they contradict  
    // each other, what I mean is  
    // abstract tells the class to override and static methods cannot be overridden since they are  
    // global to class  
  
    private int x;  
  
    public Parent(int x) {  
        this.x = x;  
    }  
  
    // public abstract Parent(); // abstract constructor are not allowed  
  
    // abstract class can have non-abstract methods (concrete methods) which can be called using  
    // subclass  
    void greeting() {  
        System.out.println("Good Morning! This is parent.");  
    }  
    public int getX() {  
        return x;  
    }  
  
    // NOTE: we cannot achieve multiple inheritance through abstract classes, why?  
    // -> bcz, abstract class still allows concrete methods, which may cause ambiguity  
}
```

```
public class Child extends Parent{  
  
    public Child(int x1) {  
        super(x1); // we can call the constructor of through child class  
    }  
  
    @Override  
    void run() {  
        System.out.println("Child is running");  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Parent child = new Child(20);  
        child.run(); // -> Child is running  
        System.out.println(child.getX()); // -> 20  
  
        // Parent parent = new Parent(); // cannot instantiate an abstract since it has undefined (only  
        // declared) methods  
  
        // But, only way to create object of parent is by overriding the methods;  
        Parent parent = new Parent(10);  
        // overriding the run method while creating the object of the parent  
        @Override  
        void run() {  
            System.out.println("Parent is running");  
        };  
        parent.run(); // -> Parent is running  
    }  
}
```

**Interface:** is a reference type that defines a contract of methods that implementing class must adhere to

→ interface can have only abstract methods.

from java 8, it can have static & default method  
also, why? → backward compatibility, (future proof)

→ variables declared are static & final.

**NOTE:** Abstract class can provide the implementation of interface.  
Interface can't provide the implementation of abstract class.

→ An interface can extend another Java interface only  
→ Members of interface are by default public

```
public class Sphere implements SolidShape {
    int r;

    public Sphere(int r) {
        this.r = r;
    }

    @Override
    public double volume() {
        return (4/3) * Shape.PI * Math.pow(this.r, 3);
    }

    @Override
    public double area() {
        return Shape.PI * Math.pow(this.r, 2);
    }
}
```

```
public interface Shape {
    // static void fun(); // static interface
    // method should always have a body

    public double area();

    double PI = Math.PI; // variable in
    // interface are by default final and static
}
```

```
// when we extend an interface then child
// interface will inherit all the parent
// abstract methods and variables

public interface SolidShape extends Shape{
    public double volume();
}
```

```
public class Circle implements Shape, Color{
    private int r;
    public Circle(int r) {
        this.r = r;
    }

    @Override
    public double area() {
        return Shape.PI * Math.pow(this.r, 2);
    }

    @Override
    public void paint(COLORS color) {
        System.out.println("Circle is painted with " +color+ " color");
    }
}
```

```
public interface Color {
    enum COLORS{
        RED,
        ORANGE
    };

    public void paint(COLORS color);
    public default void greet() {
        System.out.println("this is color interface");
    }
}
```

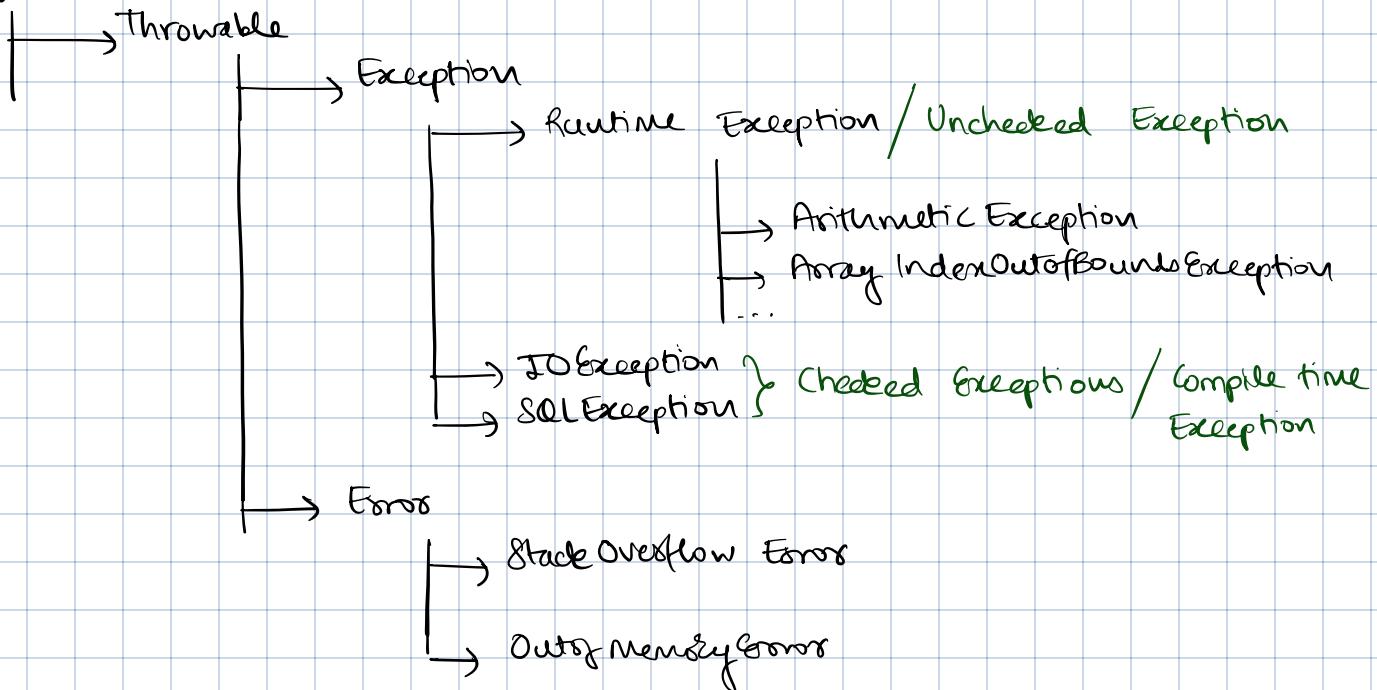
```
public class Main {
    public static void main(String[] args) {
        Circle circle = new Circle(10);
        System.out.println((circle.area())); // -> 314.1592653589793
        circle.paint(Color.COLORS.RED); // -> Circle is painted with RED color

        SolidShape sphere = new Sphere(15);
        System.out.println((sphere.area())); // -> 706.8583470577034
        System.out.println((sphere.volume())); // -> 10602.875205865552
    }
}
```

## Exception Handling

is an event that disrupts the normal flow of a program's execution.

Object



Java exception handling is managed via five keyword:

1. try: Program statement that you think can raise exception are contained within a try block.  
ex. file handling
2. Catch: the exception occurred within try block are thrown, and to handle those we have catch block.
3. finally: the code which will be executed irrespective of whether the exception is thrown or not.  
ex. closing the file.
4. Throw: This is used to manually generate some exceptions.
5. throws: any exception that is thrown out of a method must be specified as such by a throws clause.

```

try {
    // block of code to monitor for errors
    // The code you think can raise an exception
}

sub class
| } catch ( ExceptionType2 ex0b ) {
|     // exception handler for ExceptionType2
| }
super class
| } catch ( ExceptionType2 ex0b ) {
|     // exception handler for ExceptionType2
| }
| } // optional
finally {
    // block of a code to be executed after try block
    ends.
}

```

Custom Exceptions: we can create our own custom exceptions by extending the 'Exception' or 'Runtime Exception' classes

```

class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}

public class CustomExceptionExample {
    public static void main(String[] args) {
        try {
            throw new CustomException("This is a custom exception.");
        } catch (CustomException e) {
            System.out.println(e.getMessage());
        }
    }
}

```

NOTE: when you perform the operation  $a/b$  where  $a$  is a double &  $b$  is 0, in floating-point arithmetic the result is considered mathematically undefined & is represented as "infinity". This is bcoz floating-point arithmetic follows IEEE 754 Standard which defines special values like positive & negative infinity to handle such cases.

on the other hand, in integer arithmetic, divide by zero is not mathematically possible & is considered an error

## Object Cloning:

- ① Copy Constructor: it is a user-defined constructor that you implement in your class. You have full control over how the copying process is implemented. You can choose whether to perform a shallow or deep copy, or any other custom copy logic you need.
- ② Cloning: The clone method is a method provided by Java runtime in the Object class. You can override it to customize the copying behavior for your class.
- If you want to achieve a deep copy, you need to implement the Cloneable interface & provide your own deep copying logic.

### SHALLOW COPY.

```
public class Product implements Cloneable{
    String name;
    int price;
    int arr[] = new int[]{1, 2, 3, 4};

    public Product(String name, int price) {
        this.name = name;
        this.price = price;
    }

    // copy constructor
    public Product(Product other) {
        this.name = other.name;
        this.price = other.price;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        // this is shallow copy
        return super.clone();
    }
}
```

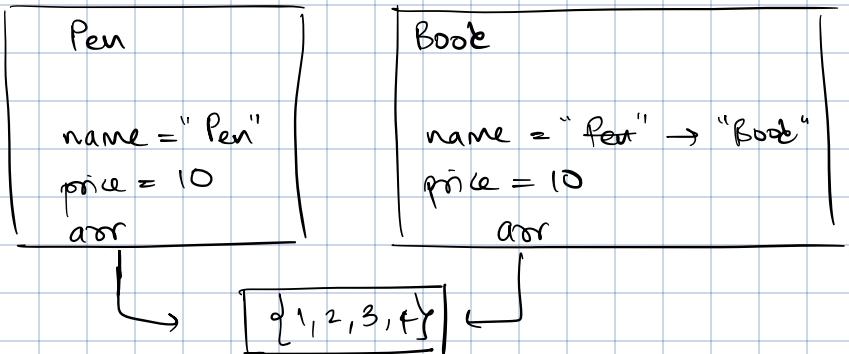
```
public class ShallowCopy {
    public static void main(String[] args) throws CloneNotSupportedException {
        Product pen = new Product("pen", 10);
        Product book = (Product) pen.clone();

        System.out.println(pen.name);
        System.out.println(book.name);

        // changing the name of pen object
        book.name = "book";

        // name attribute changed in book object and it has not reflected in pen object
        System.out.println(Arrays.toString(pen.arr));
        System.out.println(Arrays.toString(book.arr));
        pen.arr[0] = 20;

        // BUT, in non-primitive data type like arr the changes in one object reflected in copied (book) object also
        // this is called SHALLOW COPY
        System.out.println(Arrays.toString(pen.arr));
        System.out.println(Arrays.toString(book.arr));
    }
}
```



both the objects were still referring the same old non-primitive member.

i.e. shallow copy does not create new instance of the non-primitive members.

## DEEP COPY

```
public class DeepProduct implements Cloneable{
    String name;
    int price;
    int arr[] = new int[]{1, 2, 3, 4};

    public DeepProduct(String name, int price) {
        this.name = name;
        this.price = price;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        // below is deep copy steps

        // 1. first we copy the object in another reference variable
        DeepProduct deepProduct = (DeepProduct) super.clone();

        deepProduct.arr = new int[this.arr.length]; // recreating new array for copied object

        // copying all the elements from old object arr
        for(int i=0; i<this.arr.length; i++) {
            deepProduct.arr[i] = this.arr[i];
        }

        return deepProduct;
    }
}
```

```
public class DeepCopy {
    public static void main(String[] args) throws CloneNotSupportedException {
        DeepProduct pen = new DeepProduct("pen", 10);
        DeepProduct book = (DeepProduct) pen.clone();

        System.out.println(pen.name);
        System.out.println(book.name);

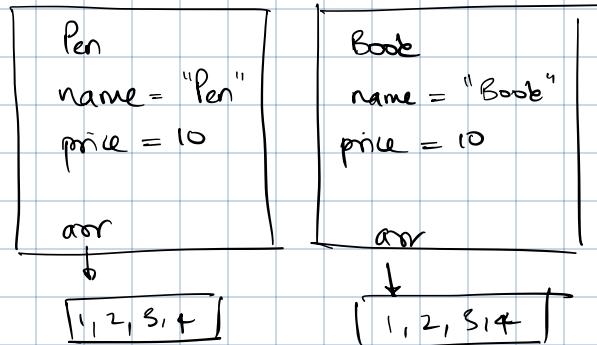
        // changing the name of pen object
        book.name = "book";

        // name attribute changed in book object and it has not reflected in pen object
        System.out.println(pen.name);
        System.out.println(book.name);

        System.out.println(Arrays.toString(pen.arr));
        System.out.println(Arrays.toString(book.arr));
        pen.arr[0] = 20;

        // Even after changing the value in arr member of pen object, there is
        // no change in the arr member of book object
        // this is DEEP COPY
        System.out.println(Arrays.toString(pen.arr));
        System.out.println(Arrays.toString(book.arr));
    }
}
```

we are first cloning the object to other new object.  
then we are creating new instance of arr to it filling the values



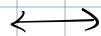
## Enums:

- it is a class type.
- we can give them constructors, instance variables & methods, & even implement interface

## Properties:

- ① An enumeration is a list of named constants (an object of type enum).
- ② Every enum constant is always implicitly public static final. Since it is static, we can access it by using the enum name. Since it is final, we can't create child enums.

```
enum Colors {
    RED,
    BLUE,
    GREEN
}
```



```
class Colors
```

```
public static final Color Red = new Color();
public static final Color Blue = new Color();
public static final Color Green = new Color();
```

NOTE: - All enums implicitly extend java.lang.Enum class. As a class can only extend one parent in Java, so an enum cannot extend anything else.

- `toString()` is overridden in `java.lang.Enum` class which return the enum constant name.

### Values(), ordinal(), & valueOf()

- `values()`: returns all values present inside enum.
- `ordinal()`: returns the index or position of the enum constant where it was declared inside enum.
- `valueOf()`: returns the enum constant of specified string value.

NOTE: - the constructor is executed separately for each enum constant at the time of enum class loading.  
 - we can't invoke invoke constructor directly.

NOTE: - Enum can contain concrete method.

- Enum can have abstract methods , but they need to be defined for each constant .

```
package l_enums;

public enum Color {
    RED(1) {
        // abstract methods are overridden and defined
        @Override
        public String greet() {
            return "Hey, this is " + this.name() + "color";
        }
    }, ORANGE(2) {
        @Override
        public String greet() {
            return "My color is"+this.name();
        }
    }, BLUE(3) {
        @Override
        public String greet() {
            return "I'm am of sky color";
        }
    };
    // enums can have variables
    int value;
    // constructor cannot be public
    // the constructor is called for all the constants present inside
    Color(int value) {
        System.out.println("Constructor for color" + value);
        this.value = value;
    }
    // enums can have concrete method
    public int getValue() {
        return value;
    }
    // enums can also have abstract methods, but then each instance of enum class must implement this method
    public abstract String greet();
}
```