Whenever there is overriding i.e. when we create object of child class then it will call function of child class only.

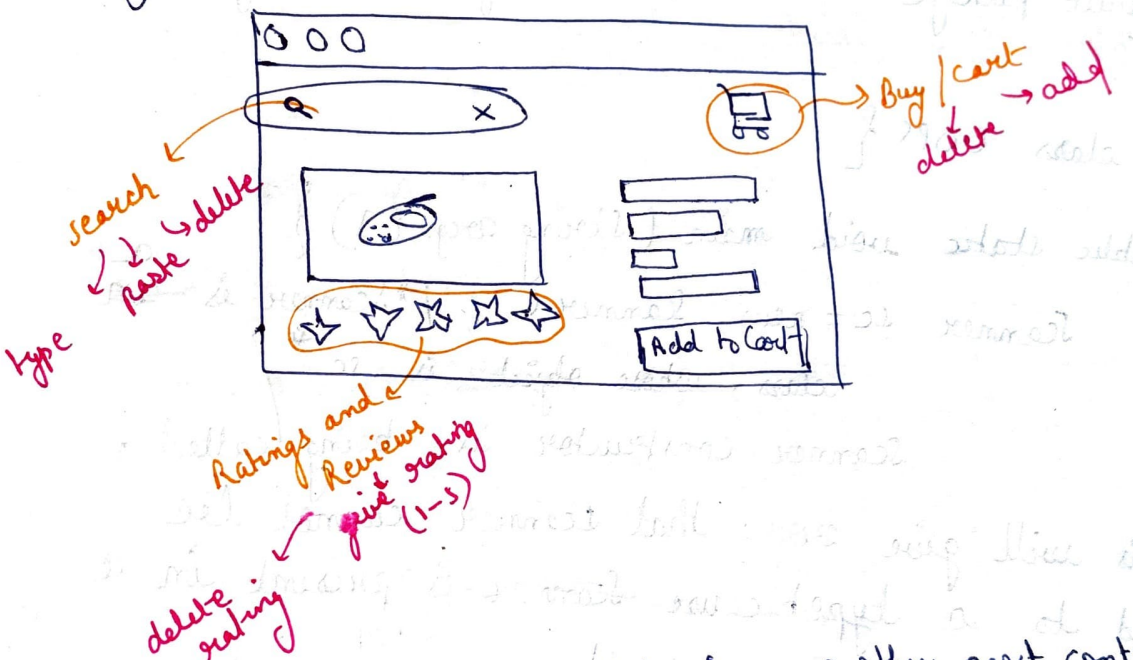
19.18

Packages

In real life we try to place similar objects in a ~~sim~~ similar place like toys in 1 box, books in 1 box for easy access.

Similarly when we write code, we group same type of functions, classes in a box (package).

Amazon.com

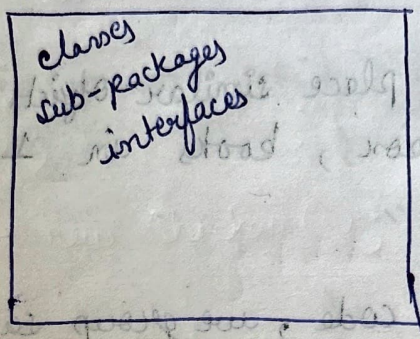


One part controls Ratings and Reviews, another part controls ~~buy~~ buy / cart, another part controls search. In Ratings and Reviews we can delete rating, give rating, in buy / cart we can delete or add ~~it~~ item.

We logically want Rating to be in same package.

Eg. Maps , search , chrome
↙ in one package
↳ in one package
↳ in one package.

Package is a group of similar types of classes, interfaces and sub-packages.



There are two types of packages in Java:
> In-built package
> User-defined package

```
public class OOPS {
    public static void main (String args[]) {
        Scanner sc = new Scanner(); /* Scanner is a class, whose object is sc */
```

Scanner constructor is being called.

But this will give error that scanner cannot be resolved to a type because Scanner is present in a package util, so we import it.

util package is a utility package that contains helpful classes, packages, interfaces.

19.19 Abstraction

Abstraction is very similar to Encapsulation. In encapsulate we encapsulate data and functions together and with the help of access modifiers/ access specifiers we hide unnecessary details. But in abstraction we just don't hide, we also show the important parts to the user.

Abstraction - Hiding all the unnecessary details and showing only the important parts to the user.

Abstraction is implemented in two ways:
- Interfaces
- Abstract Classes

Whenever we do abstraction in programming, we give idea to the user that we have to implement in a particular way, but we don't give the actual implementation.

idea ✓

implementation ✗

19.20 Abstract Classes

To make a class Abstract we use Abstract keyword.

```
abstract class A {

}
```

When a class is Abstract it has the following properties
1. Cannot create an instance (object) of abstract class.
2. Can have abstract/ non-abstract methods.

eg. abstract method →

```
abstract void fun ();
```

}

3. Can have constructors.

```
public class OOPS {
    public static void main (String args []) {
        Horse h = new Horse();
        h.eat();
        h.walk();
        Chicken c = new Chicken();
        c.eat();
        c.walk();
    }
abstract class Animal {
    void eat() {
        System.out.println("animal eats");
    }
    abstract void walk(); // abstract method
```

It is an abstract method as we have not written implementation of this method.

Every animal has walk() method but how does it look will not be demonstrated by Animal, it will be given by subclasses itself.

```
class Horse extends Animal { // Now it is compulsory for Horse
                             to implement walk() method.
    void walk() {
        System.out.println("walks on 4 legs");
    }
}
```

```
class Chicken extends Animal {
    void walk() {
        System.out.println("walks on 2 legs");
    }
}
```

o/p : animal eats
walk on 4 legs
animal eats
walk on 2 legs.

We can have constructors in Abstract class. Let us create constructor for Animal class.

```
abstract class Animal {
    String color;
    Animal() {
        color = "brown";
    }
    void eat() {
        s.o.pln("animal eats");
    }
    abstract void walk();
}
class Horse extends Animal {
    void changeColor() {
        color = "dark brown";
    }
```

```
    void walk(){
        S.o.pln("walks on 4 legs");
    }
}
class Chicken extends Animal{
    void changeColor(){
        color = "Yellow";
    }
    void walk(){
        System.out.println("walk on 2 legs");
    }
}
```

Whenever horse object is created then by default horse will be created of brown color until and unless we call changeColor().

```
public static void main(String args[]){
    Horse h = new Horse();
    h.eat();
    h.walk();
    S.o.pln(h.color);
}
```

o/p : animal eats
walk on 4 legs
brown

Whenever we create object of child class then constructor of parent is called first.

Constructor's task can also be to for child class's object's variables to be initialised.

Property's initialisation follows from top to bottom in inheritance hierarchy.

```
public class OOPS {
    public static void main (String args [] ) {
        Mustang myHorse = new Mustang();
    }
}
abstract class Animal {
    String color;
    Animal () {
        System.out.println ("animal constructor called");
    }
    void eat () {
        System.out.println ("animal eats");
    }
    abstract void walk();
}
class Horse extends Animal {
    Horse() {
        So.pln (" Horse constructor called ");
    }
    void changeColor() {
        color = "dark brown";
    }
```

```
    void walk () {
        System.out.println ("walks on 4 legs");
    }
}
class Mustang extends Horse {
    Mustang () {
        S.o.pln ("Mustang constructor called");
    }
}
```

Hierarchy:

Animal
↓
Horse
↓
Mustang

o/p :

animal constructor called
Horse constructor called
Mustang constructor called

19.21 Interfaces

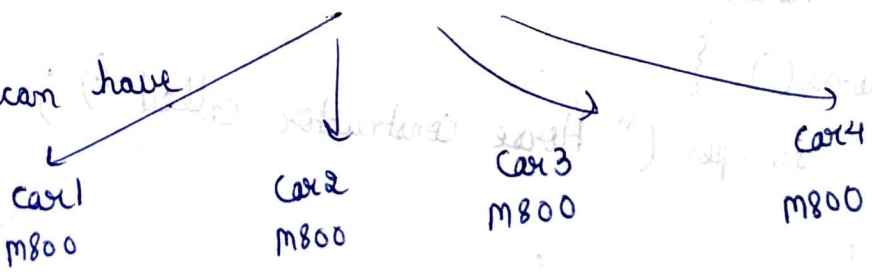
Interface is a blueprint of ~~class~~ class.

Example, we can have interface car (wheels, speed, engine)

↓

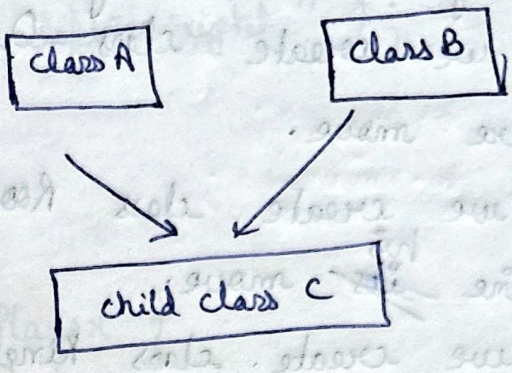
M800 (class)

Then we can create various objects.



• In Interfaces implements multiple ~~inheritance~~ inheritance.

Multiple inheritance :



• Interfaces implements total - abstraction.

In Animal class we saw that we had two functions, eat and walk in which walk was abstract but eat was not abstract. This is not 100% abstraction. But in interfaces there is always 100% abstraction.

Properties :

- All methods are public, abstract and without implentation
- Used to achieve total abstraction.
- Variables in the interface are final, public and static.

For interface we use Interface keyword.
To inherit interface we write implements.

Interface → implements

class → extends

Let us say we want to ~~play~~ play chess:

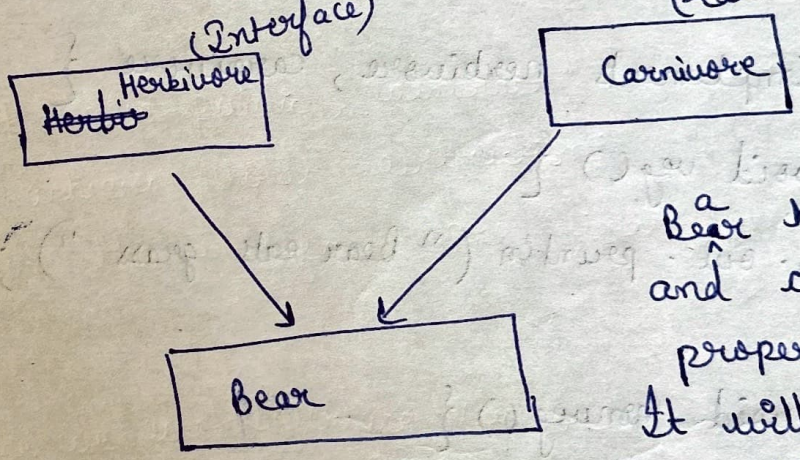
Chessplayer has property moves.

- If we create class Queen then we can define here move.
- If we create class Rook (elephant) then we can define his ~~its~~ move.
- If we create class King then we can define his move.

```
public class OOPS {
    public static void main (String args[]) {
        Queen q = new Queen ();
        q.moves ();
    }
}
interface ChessPlayer {
    void moves (); // Each chessPlayer should have a function which tells about its moves. It is public, abstract and we can write its implementation. Player */
}
/* Let us make classes for chess~~Play~~
class Queen implements ChessPlayer {
public void moves() {    //we write keyword public otherwise it will be default
    S.o.pln ("up, down, left, right, diagonal (in all four direc^ns)");
}
}
```

```
class Rook implements ChessPlayer {
    public void moves() {
        s.o.pln("up, down, left, right");
    }
}
class King implements ChessPlayer {
    public void moves() {
        s.o.pln("up, down, left, right, diagonal");
    }
}
```

Let us see how to implement ~~multiple~~ multiple inheritance using interface:



a Bear has both herbivore and carnivore properties.
It will inherit both properties.

Herbivore, Carnivore gives idea about their properties.
Function will be implemented by Bear class.

```
public class OOPS{
    public static void main (String args [ ]) {
        Bear b = new Bear ( );
        b.veg ( );
        b.nonveg ( );
    }
}
interface herbivore {
    void veg ( );
}
interface carnivore {
    void nonveg ( );
}
class Bear implements herbivore, carnivore {
    public void veg ( ) {
        System.out.println ( " Bear eats grass " );
    }
    public void nonveg ( ) {
        System.out.println ( " Bear eats meat " );
    }
```