Agenda :-

- Constructor
- Types of Constructor
- Deep Copy & Shallow Copy
- Inheritance
- Polymorphism
- Method Overloading & Method Overriding

Constructors

Class:- Blueprint of an entity.

Object:- Instance of a class.

class
```
Student {
    String name;
    int age;
    doubly psp;
}
```

Student st = new Student();

int a = 12;

→ 10x.

↓ Constructor

default constructor

Heap

10x

name = null
age = 0
psp = 0.0

If we don't create our own constructor in a class, a **default constructor** is created.

Default constructor creates a new object of the class and sets the value of each attribute as the default value of that type.

Examples of default values of datatype:

- **0** for integer,
- **null** for String,
- **0.0** for float, etc.

```
class Student {
    String name;
    int age;
    double psp;
    String univName;

    Student() {
        name = null;
        age = 0;
        psp = 0.0;
        univName = null;
    }
}
```

→ don't create it.

→ Constructor name is same as class Name

→ if we create our own constructor, then default constructor will not get created

constructor is a function which is called automatically when the object is created, its name will be same name as of the class name and it has no return type

Summarising the default constructor:

1. Takes no parameter.
2. Sets every attribute of class to it's default value (unless defined).
3. Created only if we don't write our own constructor.

* Manual Constructor :-

```
public class Student {
    String name;
    private int age [redacted];
    String univName;
    double psp;

    public Student (String studentName, String universityName) {
 this. name = studentName;
 this. univName = universityName;
    }
}
```

Student st = new Student ("Puneet", "Harvard");

Student st2 = new Student (); // error

name = puneet
age = 0
univname = Harvard
psp = 0.0

But here, why its throwing error?

- Because now there is no default constructor, since we have our own constructor, and it has parameters. So we have to pass the parameters here.

```java
public class Student {
    String name;
    private int age [redacted]
    String univName;
    double psp;

    public Student (String studentName, String universityName) {
this. name = studentName;
this. univName = universityName;
    }
}
```

Public Student ( ) {

}

Public Student (String name, int age, String univName)

}

Student st = new Student ("Puneet", "Harvard");

Student st2 = new Student ();

Copy Constructor

name PSP

Student st1 = new Student ("Sagar", 20);

Student st2 = st1;

st2 = 1ak

st1 =

1ak

name = Sagar
PSP = 20

```
class Student {
    Private String name;
    Private int age;

    Student() {
        name = null;
        age = 0;
    }
    Student(Student st) {
        name = st.name;
        age = st.age;
    }
}
```

Copy Constructor.

1ak

name = Vikas
age = 25

Student st1 = new Student ();
st1.name = "Vikas";
st1.age = 25;

1ak

Student st2 = new Student (st1);

2ak

name = Vikas
Age = 25

- The copy constructor comes in use when we have an object, and a newly created object needs the same values, so we don't assign it ourselves. We use the copy constructor the get the work done.
- Some of the attributes may be private and cannot be accessed by the user, but a copy constructor can access it and make the copy itself.

Student st2 = st1; // NO.

this is just a new reference pointing to the same object so if I do any change via ST2, that will get reflected in ST1 as well

Deep Copy & Shallow Copy

Shallow copy

- When we have created a new object, but behind the scenes, the new object still refer to attributes of the old object. i.e., the new object still refers to the same data as the old copy.

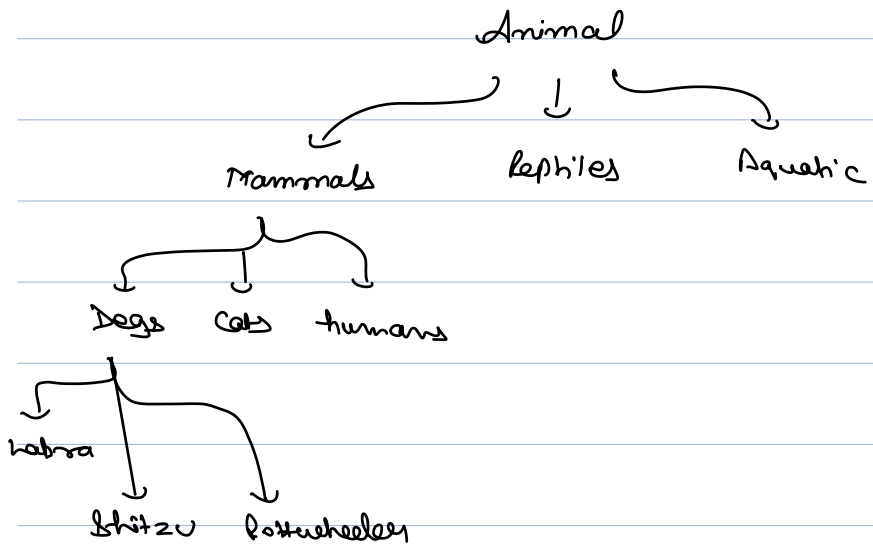
```
original_books = ["Book A", "Book B"]
shallow_copy_books = original_books

shallow_copy_books.append("Book C")

print(original_books)  # Output: ["Book A", "Book B", "Book C"]
```

Deep copy

- When we have created a new object behind the scenes, the new object do not refer to attributes of the old object. i.e., the new object has no shared data.

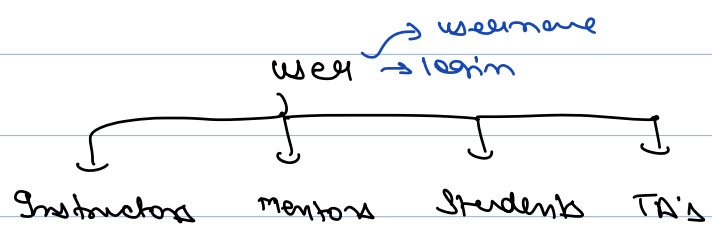
```
import copy

original_books = ["Book A", "Book B"]
deep_copy_books = copy.deepcopy(original_books)

deep_copy_books.append("Book C")

print(original_books)  # Output: ["Book A", "Book B"]
```

Inheritance



Representation of hierarchies in classes is known as inheritance.

Scaler's hierarchy



A dog can bark
but not all animals bark.

So, a **child class / subclass** can have specific attributes / behaviors which may not be present in the **parent class / superclass**.

So we can **conclude**: A child class inherits all the members of the parent class and may or may not add their own members.

ex Package a';

```
class User {
    String userName;

    void login() {
        ...
    }
}
```

class Instructor extends User.

↳ extend (in java)

- **In Python:** The inheritance is indicated using **parentheses**.
  For example: `class Subclass(SuperClass):`
- **In C++:** The inheritance is specified using a **colon**.
  For example: `class Subclass : public SuperClass { };`
- **In C#:** The inheritance is specified using a **colon**.
  For example: `class Subclass : BaseClass { }`

Package a',

```
class Instructor extends User {
    String batchName;
    double avgRating;

    void scheduleClass() {
        ...
    }
}
```

- Does the Instructor class needs a username property?
  - Yes
- Do we need to code it?
  - No
- So, how can we use it?
  - We are extending it from the User class.

Extends means, **keeping the original things and adding more things to it**.

Constructor chaining :-

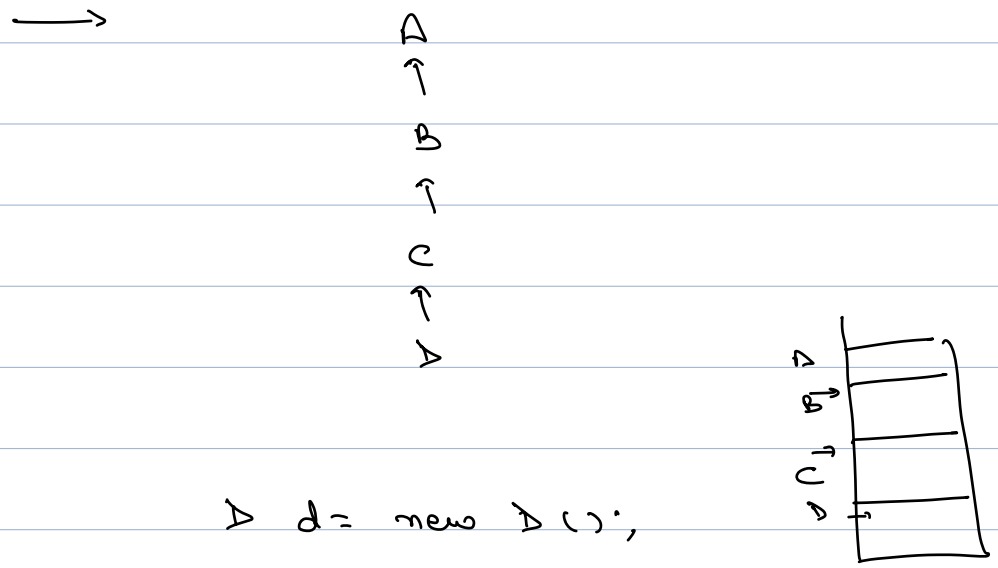
```
Instructor i = new Instructor();
    i.username = 'Aman';
    i.login();
    i.avgRating = 4.8;
```

username = ~~null~~ Aman
batchName = null
avg. Rating = 0.

A
B
C
D
E

Note: In Inheritance, a parent class is nothing but generalization, and every child is a specification.

(Assuiming no constructor is created, its only default constructors are present)



So, **What really happens when we call `D()`?**

1. Constructor of D will be called.
2. Since D is also a child of someone, so before its execution, it will call the constructor of C.
3. Similarly, C will call the constructor of B first.
4. And B will call the constructor of A before it's execution.

So, **Who's constructor will be finished first?**

- **A**'s constructor will be finished first, then **B** will be finished, then **C** will be finished, then **D** will be finished.

- **Can a child be born before its parents are born?**
  No, right?
  - That's why the parent class constructor will be called first. We haven't specifically called the parents constructor but by default, the parent constructor is called.

A
↑
B
↑
C
↑
D

```
public class C extends B {
    C() {
        System.out.println("Constructor of C");
    }
    C(String a) {
        System.out.println("Constructor of C with params");
    }
}
```

D d = new D();

A
↑
B
↑
C
↑
D

another situation

```
public class C extends B {
    C() {
        System.out.println("Constructor of C");
    }
    C(String a) {
        System.out.println("Constructor of C with params");
    }
}
```

```
public class D extends C {
    D() {
        super ("Hello"); // This must be the first line
        System.out.println("Constructor of D");
    }
}
```

D d = new D();

The **super("Hello");** will make the parametrized constructor from Class C become active.

Note: This super() line in the code must be written in the **first line inside a constructor**. Otherwise, it throws an error.

class A {

}

default

B extends A {

}

default

C extends B {

C ( ) {

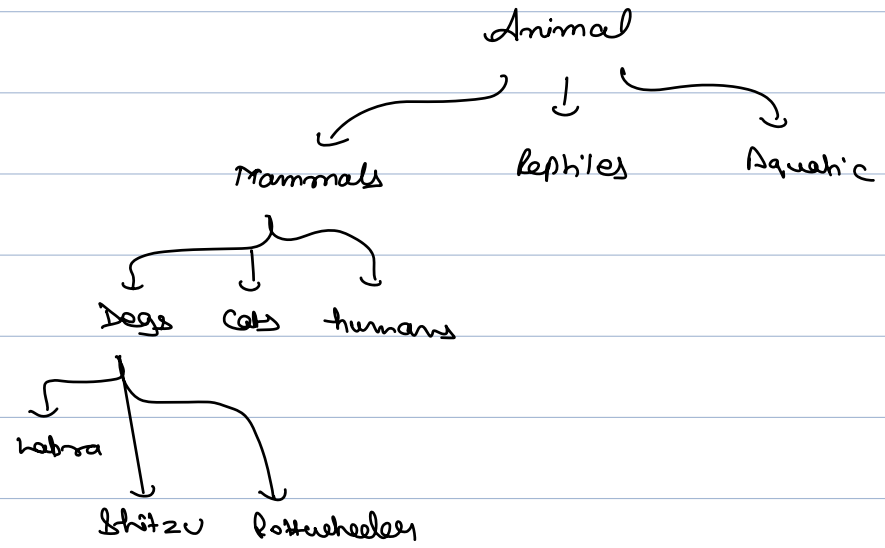
}

}

C c = new C (x, y);

Polymorphism

Poly → many
morphism → forms.



Can we write ?

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

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

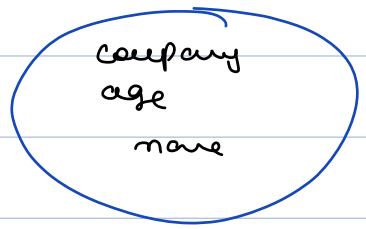
```
A {
    int age;
    String name
}
```

```
B extends A {
    String univ
}
```

```
C extends A {
    String company
}
```

```
A a = new C();
a.age =
a.name =
a.company
```

↳ compile time error.




```
C c = new A();
c.company = ?
```

- Compiler only allows you access to members of the data type of the variable.

```
Bike bike = new RoyalEnfield();
```

Two types of Polymorphism :-

1) Compile time Polymorphism
2) Run time Polymorphism.

Method Overloading

```
class A {
    void hello () {
        ___
        ___
    }

    void hello ( String name ) {
        ___
    }
}
```

→ Method Overloading :- different function having same name. (Compile Time Polymorphism)

```
hello ( );
hello ("Test");
```

Q1)

void PrintHello ( ) → PrintHello( )

void PrintHello (String s) → PrintHello (String)

Q2) void PrintHello (String s) → PrintHello (String)

void PrintHello (Integer x) → PrintHello (Int)

Q3 void PrintHello (String s) → PrintHello(String)

int PrintHello (String s) → PrintHello(String)

Method Signature.

Name of Method (Data type of Params)

Methods are known to be overloaded when they have the same name but different signatures.

PH(String)

```
void printHello(String s) {...}

String printHello(String s) {...}
```

PH(String)

Method Overriding.

```
Class A {
    void doSomething(String a) {
        ...
    }
}
```

```
Class B extends A {
    String doSomething(String c) {
        ...
    }
}
```

```
Class B extends A {
    String doSomething(String c) {
        ...
    }
    // Parent method inherited
    void doSomething(String a) {
        ...
    }
}
```

→ not allowed.

And, in the child class, we are having 2 methods with the same signature which it's not allowed. It's going to give us a **Compile time error**.

Situation - 2

```
Class A {
    void doSomething(String a) {
        ...
    }
}
```

A a = new B();

a.doSomething();

↓

Runtime Polymorphism

```
Class B extends A {
    void doSomething(String c) {
        ...
    }
}
```

If parent and child classes have the same method with the same name and same return type, and the same signature, this is called **Method overriding.**
In Method overriding, the Parent class methods get hidden.

```
class A {
    void doSomething () {
        Print (Hello)
    }
}
```

```
A a = new B();
a.doSomething()
```

O/P → Bye.

```
class B extends A
    void doSomething () {
        Print ( Bye )
    }
}
```

- The method that is executed is of the data type that is **actually** present at the time of code and **not the type of variable**.
- Do we know the exact code that is about to run in compile time?
  - No, and that's why it's called **RunTime polymorphism**