Constructors

Class → Blueprint of an idea

Object → Instance of class

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

```
int a = 10;

Student st = new Student();
```

constructor that creates object

not defined ⇒ default constructor

a) Automatically created if no constructor is defined.

b) Assign default values to variables.

int → 0

double → 0.0

String → null

boolean → false

char → ''

```
class Student {
    String name;
    int age;
    Student () {
        name = null;
        age = 0;
    }
```

public

So we have default constructor if our own constructor is created?

No

```
}
```

```
class Student {
    string name;
    int age;
    Student () {
        name = "Vikhor";
        age = 25;
    }
}
```

```
Student st = new Student();
```

st.name → "Vikhor"

st.age → 25

```
class Student {
    string name;
    int age;
    Student (string a, int b) {
        name = a;
        age = b;
    }
}
```

```
Student st = new Student();
```

Error!

Copy constructor

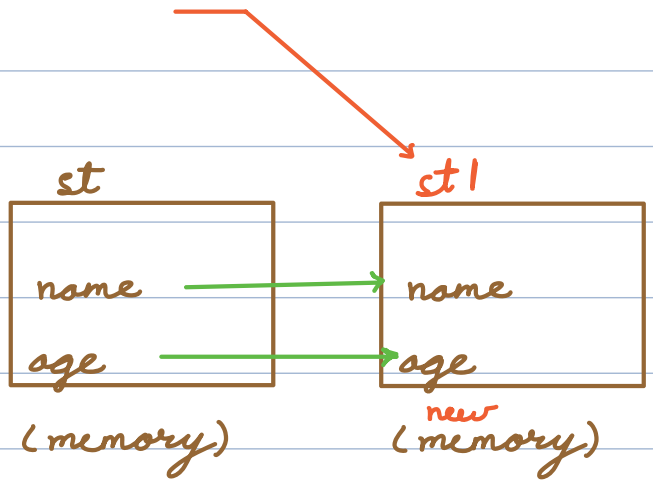
```
class Student {
    string name;
    int age;
    Student () {
        name = null;   age = 0;
    }
```

```
Student st = new Student();
st.name = "Parth";
st.age = 28;
Student st1 = new Student(st);
```

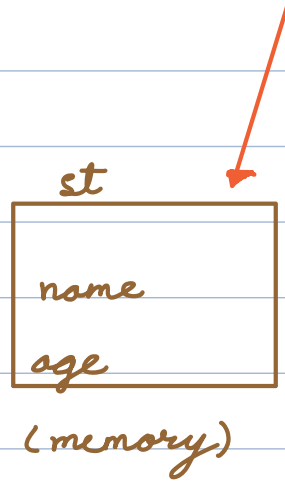
```
    Student (Student st) {  // copy constructor
        name = st.name;
        age = st.age;
    }
}
```

Is it same as → Student st1 = st No

Student st1 = new Student (st);



Student st1 = st



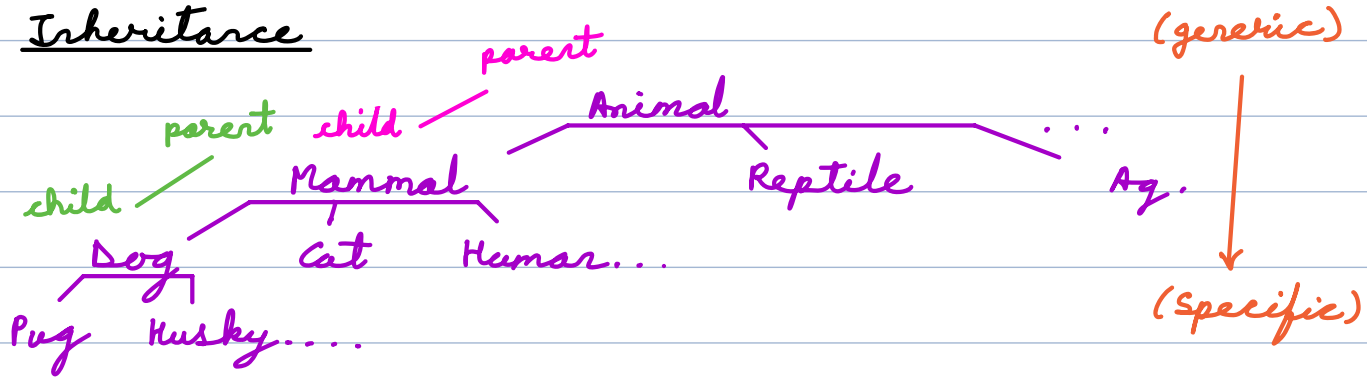
Deep Copy

Totally new memory is created for every copy. Any change in one will not impact others.

Shallow Copy

Multiple references to the same memory. Any change in one will reflect in all.

Inheritance



if (Animal can move)

⇒ Husky can move ? Yes

Animal ⇄ Mammal ⇄ Dog ⇄ Husky

Representation of hierarchies in classes → Inheritance.

parent — child relations

if (Dog can bark)

⇒ Animal can bark ? No

A child class/subclass can have specific attributes/behaviors which may not be present in parent class/superclass.

In Code →

1) Java

```
                child          parent
class Student extends User {
        ⋮
}
```

2) Python

```
class Student (User):
```

3) C++

```
class Student : public User {...};
```

4) C#

```
class Student : User {...}
```

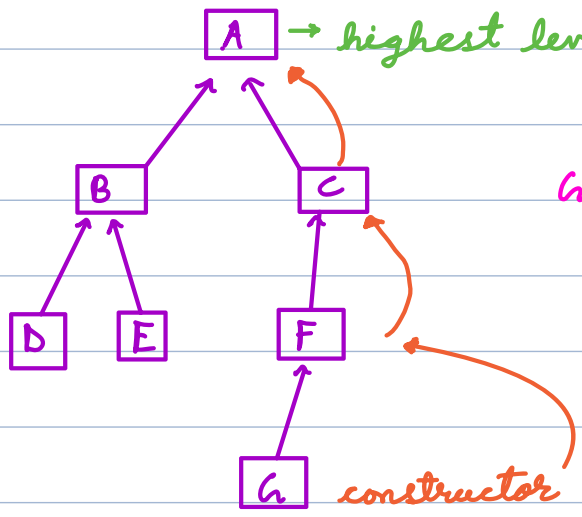
⋮

Do a child class need parent class properties to be defined? No

Extend parent attributes/methods & add more.

Constructor chaining

```
Student st = new Student();
class Student extends User {...}
```



→ highest level of abstraction

```
G x = new G();
```

constructor of G is called (4th)
→ " of F " " (" 3rd)
→ " of C " " (complete 2nd).
→ " of A " " (complete 1st) ✓

If we want to call parameterised constructor for F →

```
G() {  // constructor of G
    super(____);  // this should be 1st line
    ⋮
}
```

Polymorphism

many forms

Animal
mammal replite . . .

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

allowed ? → Yes

Dog is a Mammal &
Mammal is an Animal.

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

allowed ? → No

every animal is not a dog.

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

```
class B extends A {
    String a;
}
```

```
class C extends A {
    String x;
}
```

```
A a = new C();
```

✓

print (a.age) ✓

print (a.x) X (error)

compiler only allow access to members of the datatype of the variable.

Array → { instructors, students, TA, . . . }

child class of user

datatype of array → <u>user</u>

Types of Polymorphism

1) compile time 2) Run time

```
class A {
    ⋮
    void hello() {
        print ("Hi");
    }
    void hello (String name) {
        print ("Hi" + name)
    }
}
```

Method overloading

multiple form for methods ⇒ polymorphism

Will compiler know which function to call?

Yes → based on the parameters.

⇒ Compile time polymorphism

```
void hello()
void hello (String name)
```
✓

```
void hello (String name)
void hello (int name)
```
✓

```
void hello (String name)
String hello (String name)
```
✗

Method signature → <name of method> (datatype & count of parameters)

void hello (String name)

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

Method overriding

```
class A {
    void hello (String name) {
        print ("Hi" + name)
    }
}

class B extends A {  // child of A
    int hello (String name) {
        print ("Hi" + name)
    }
    void hello (String name) {
        print ("Hi" + name)
    }
}
```

different return type & same signature ⇒ not allowed

```
class C extends A {  // child of A
    void hello (String name) {
        print (" Hello " + name)
    }
```

overriding parent class function.

}

same method signature & return type ✓

Method Overriding

Runtime polymorphism

```
A a = new A();
a.hello("Tarif") → Hi Tarif

C c = new C();
c.hello("Tarif") → Hello Tarif
```