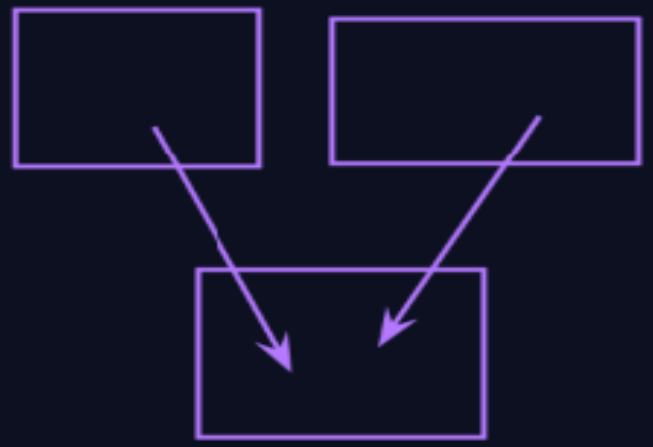
Object Oriented Programming with Java (OOPJ)

Session 5: Arrays

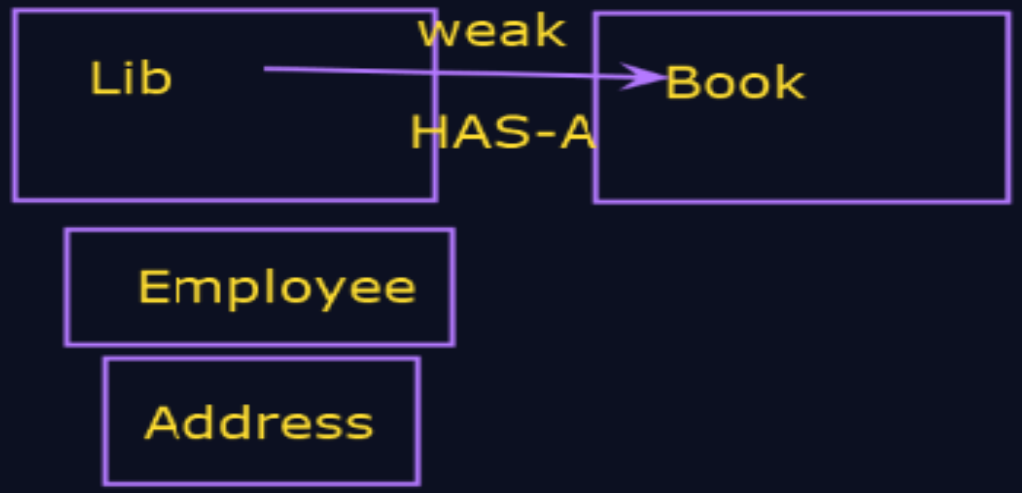
Kiran Waghmare

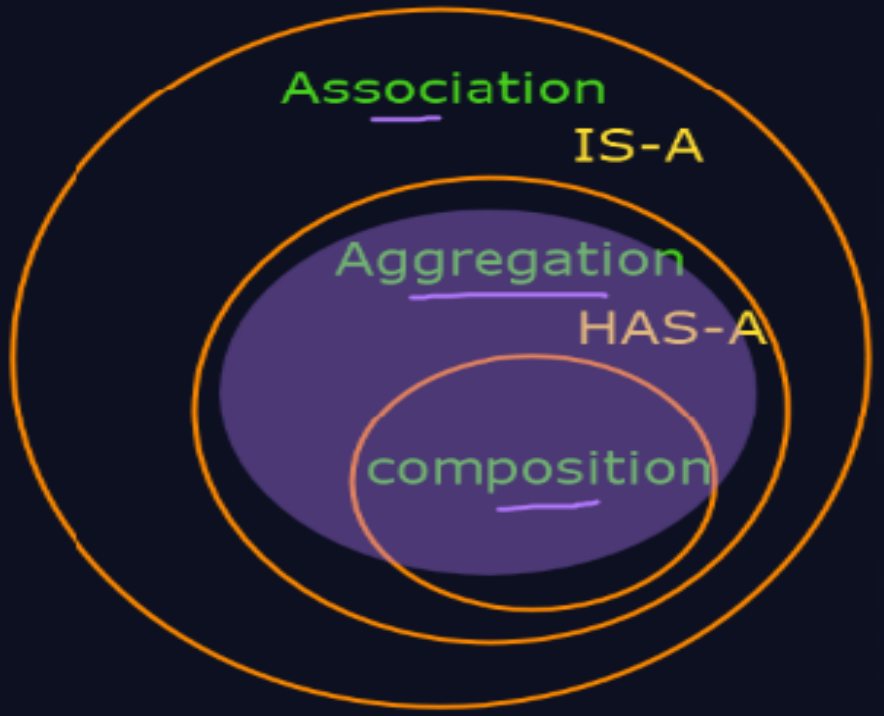
```
    }
}
```

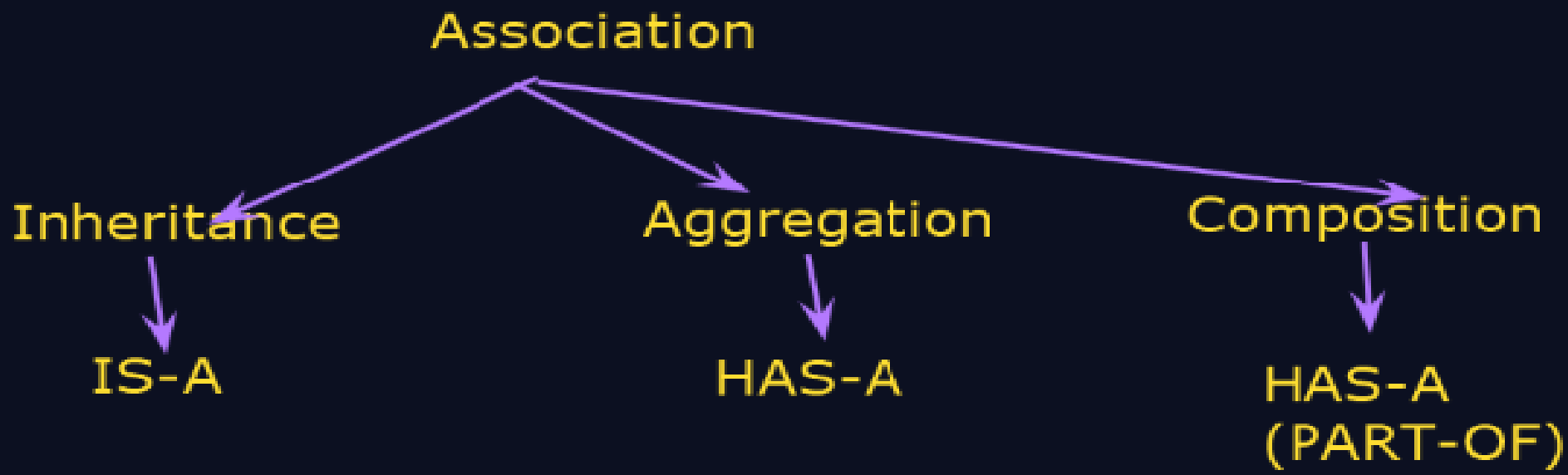



p
c


A
In
Xyz
start()


Lib
weak
HAS-A
Book
Employee
Address


Association
IS-A
Aggregation
HAS-A
composition


Association
Inheritance
Aggregation
Composition
IS-A
HAS-A
HAS-A
(PART-OF)

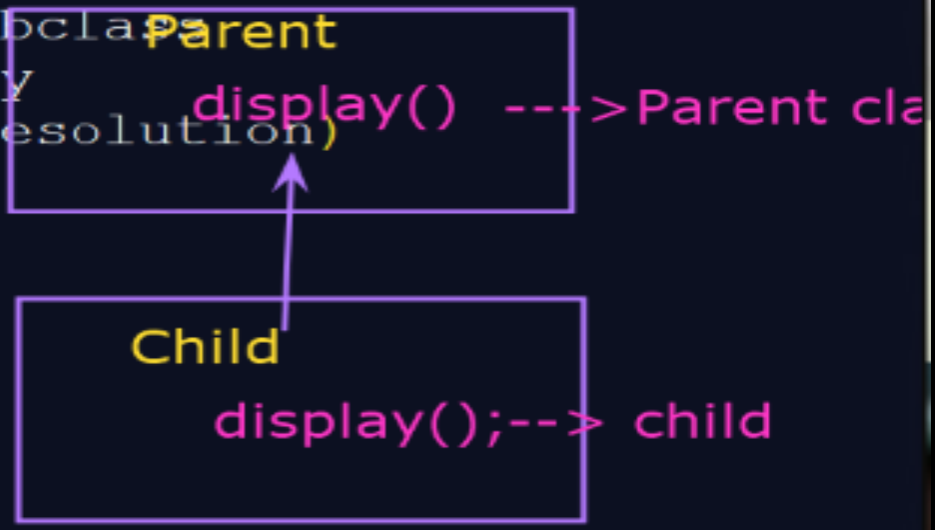
```
class Employee{
    int id;
    String name;
    Address a;//HAS-A relationship with Address class

    Employee(int id, String name, Addres a)
    {
        this.id = id;
        this.name = name;
        this.a = a;
    }

    void display()
    {
        System.out.println(id+" "+name);
        System.out.println(a.city+" "+a.state+" "+a.country);
    }

}

class Address{
    String city,state,country;
```

```
    String type;

    Engine(String type){
        this.type = type;
    }
}

class Car{
    String model;
    Engine e;//Composition (Strong HAS-A Relationship)

    Car(String model, String type)
    {
        this.model =model;
        this.e = new Engine(type);//Creating an object inside the
    }

    void display()
    {
        System.out.println(model+" "+e.type);
    }
}

class CompositionDemo{
```

```
}

class Car{
    String model;
    Engine e;//Composition (Strong HAS-A Relationship)

    Car(String model, String type)
    {
        this.model =model;
        this.e = new Engine(type);//Creating an object inside the
    }

    void display()
    {
        System.out.println(model+" "+e.type);
    }
}

class CompositionDemo{
    public static void main(String args[]){

        Car c1 = new Car("BMW","X6");
        c1.display();
```

```java
class Engine{
    String type;

    Engine(String type){
        this.type = type;
    }
}

class Car{
    String model;
    Engine e;//Composition (Strong HAS-A Relationship)

    Car(String model, String type)
    {
        this.model =model;
        this.e = new Engine(type);//Creating an object inside the constructor
    }

    void display()
    {
        System.out.println(model+" "+e.type);
    }
}

class CompositionDemo{
    public static void main(String args[]){

        Car c1 = new Car("BMW","X6");
        c1.display();
```

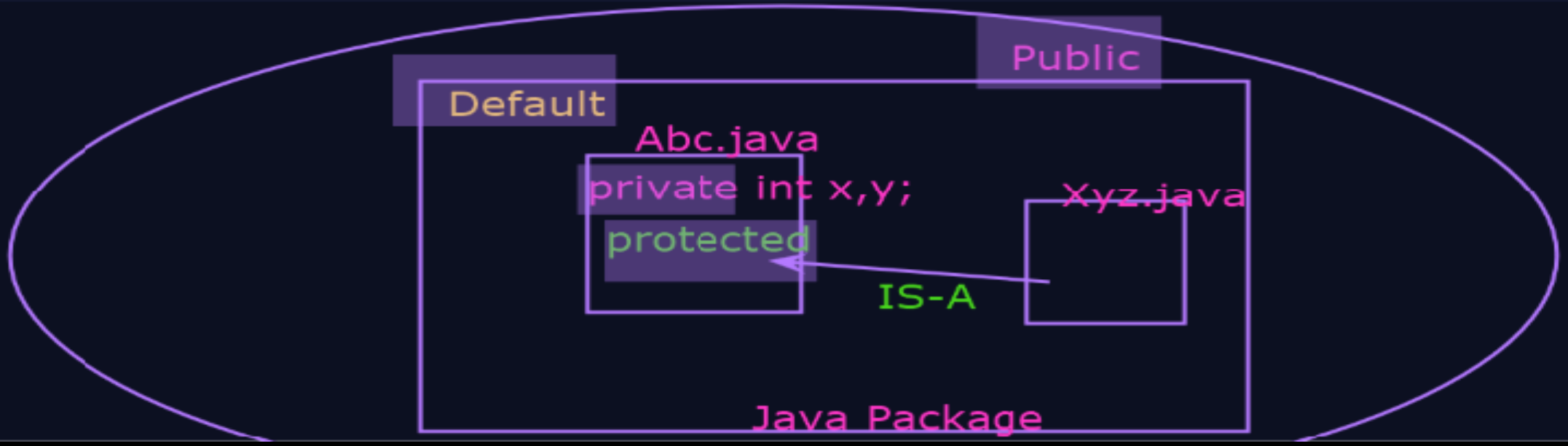
its superclass, it is called method overriding.
-Allows runtime polymorphism
-provides specific implementation in the subclass
-enhance the modularity and code reusability
-supports dynamic method dispatch(runtime resolution)



Rules for Method overriding:
---

1. Same method Name
2. Same parameters
3. IS-A relations (extends)
4. Same return type

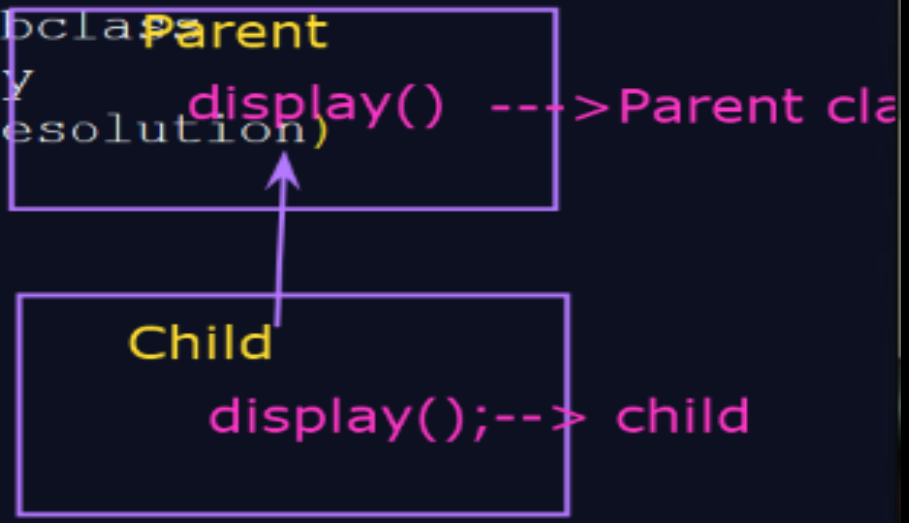
Access Modifier: public, Protected, default, Private



its superclass, it is called method overriding.
-Allows runtime polymorphism
-provides specific implementation in the subclass
-enhance the modularity and code reusability
-supports dynamic method dispatch(runtime resolution)

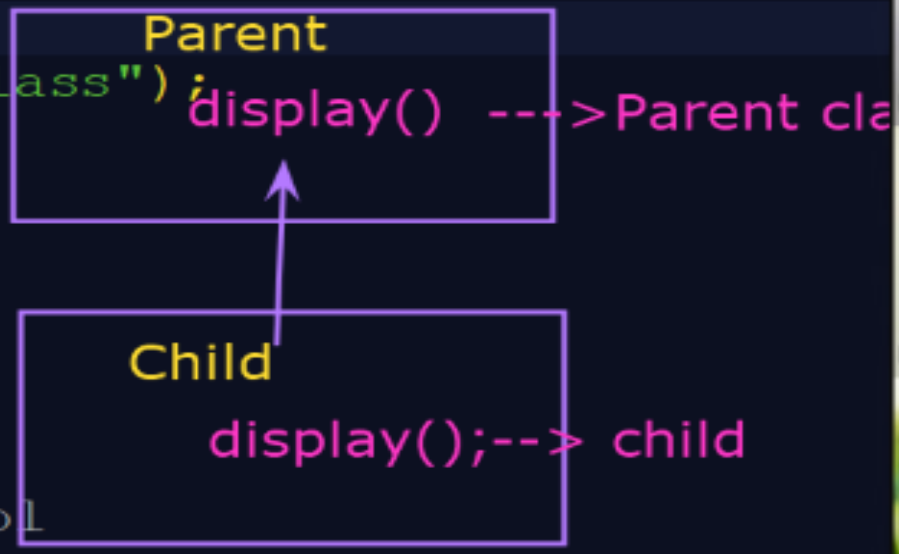
Rules for Method overriding:
---

1. Same method Name
2. Same parameters
3. IS-A relations (extends)
4. Same return type



Access Modifier: public, Protected, default, Private
-Public: class, Package, Subclass, Global
-Protected: class, Packages, Subclass
-Default: class, Package
-Private: class

```java
class Parent{
    protected void display(){
    System.out.println("Display()::Parent class");
    }
}

class Child extends Parent{
    @Override
    //protected or public
    void display(){// Method Overrriding
    //default: Error: Access modifier control

    System.out.println("Display()::Child class");
    }
}

class OverridingDemo3{
    public static void main(String args[]){

        Child c = new Child();
        c.display();
    }
}
```



	ACCESS LEVELS			
MODIFIER	Class	Package	Subclass	Everywhere
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N