3 Fundamentals of OOP in Java

CSE-220, Jul’17

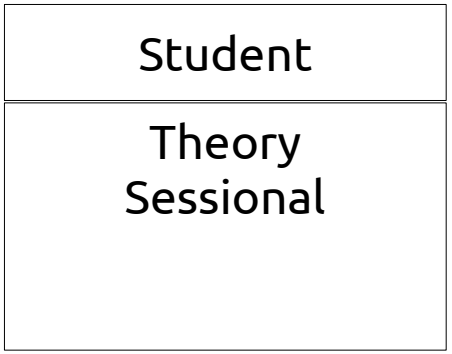
Md. Saidul Hoque Anik
Lecturer
Dept of CSE, MIST

Sthing from last class: Constructor Calling another Constructor

```
class Student
{
    int roll, term1, term2, term3;
    Student()
    {
        this(0, 0, 0, 0);
    }
    Student(int roll)
    {
        this(roll, 0, 0, 0);
    }
    Student(int roll, int term1, int term2, int term3)
    {
        this.roll = roll;
        this.term1 = term1;
        this.term2 = term2;
        this.term3 = term3;
    }

}
```

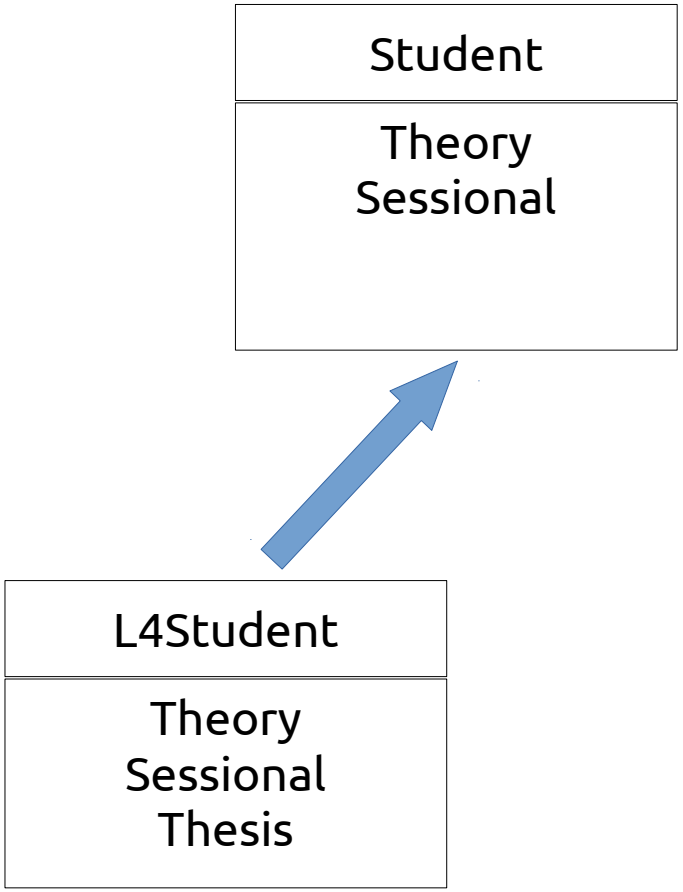
(1/3) Inheritance

Going from general to specific



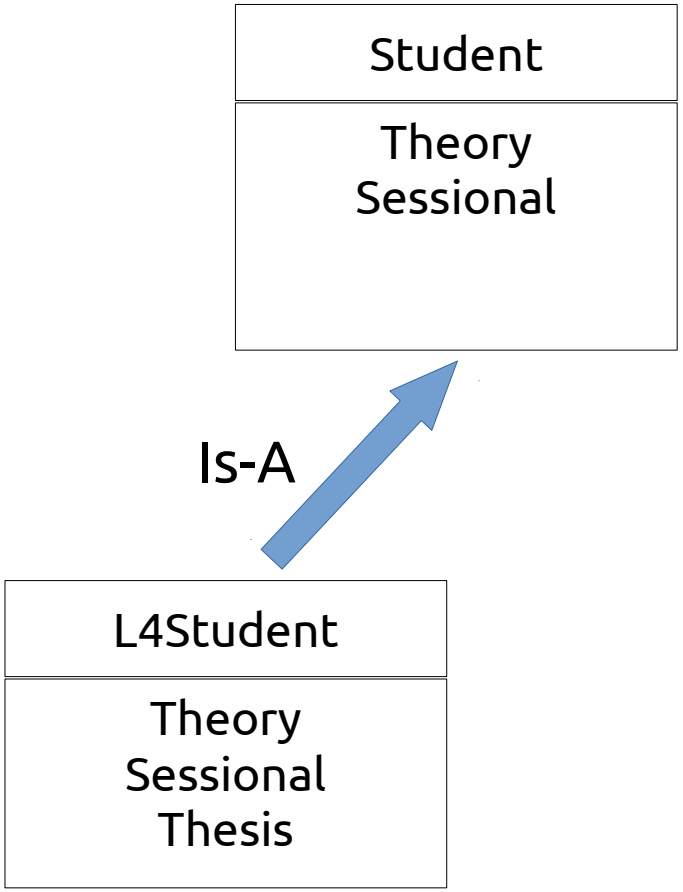
(1/3) Inheritance

Going from general to specific



(1/3) Inheritance

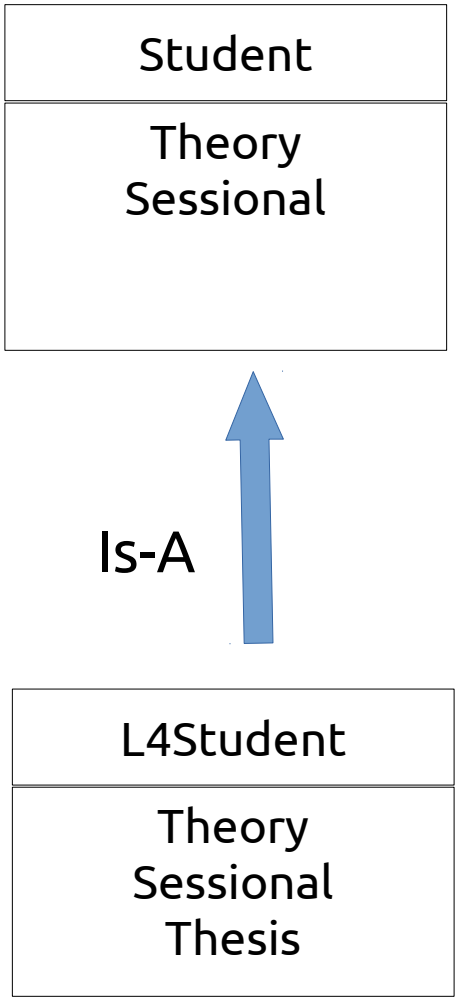
Going from general to specific



(1/3) Inheritance

Constructor

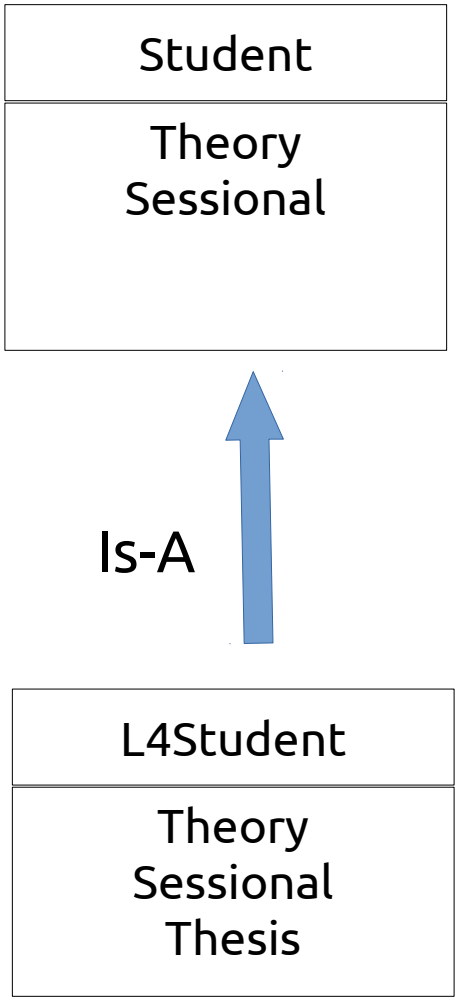
```
class Student
{
    int theory, sessional;
    public Student(int theory, int sessional)
    {
        this.theory = theory;
        this.sessional = sessional;
    }
}
class L4Student extend Student
{
    int thesis;



}
```



(1/3) Inheritance

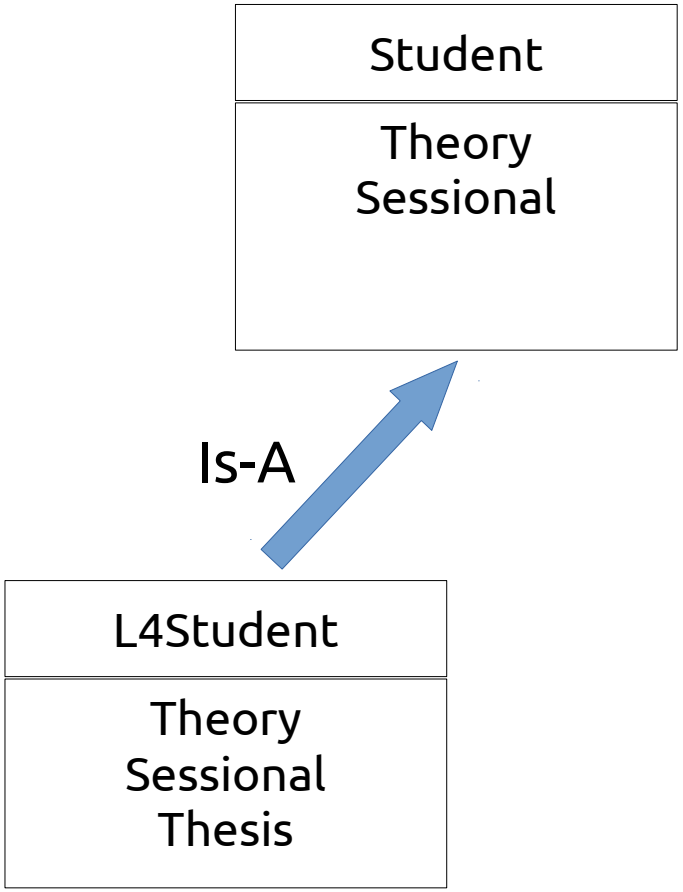
Super Constructor

```
class Student
{
    int theory, sessional;
    public Student(int theory, int sessional)
    {
        this.theory = theory;
        this.sessional = sessional;
    }
}
class L4Student extends Student
{
    int thesis;
    public L4Student(int theory, int sessional,
            int thesis)
    {
        super(theory, sessional);
        this.thesis = thesis;
    }
}
```



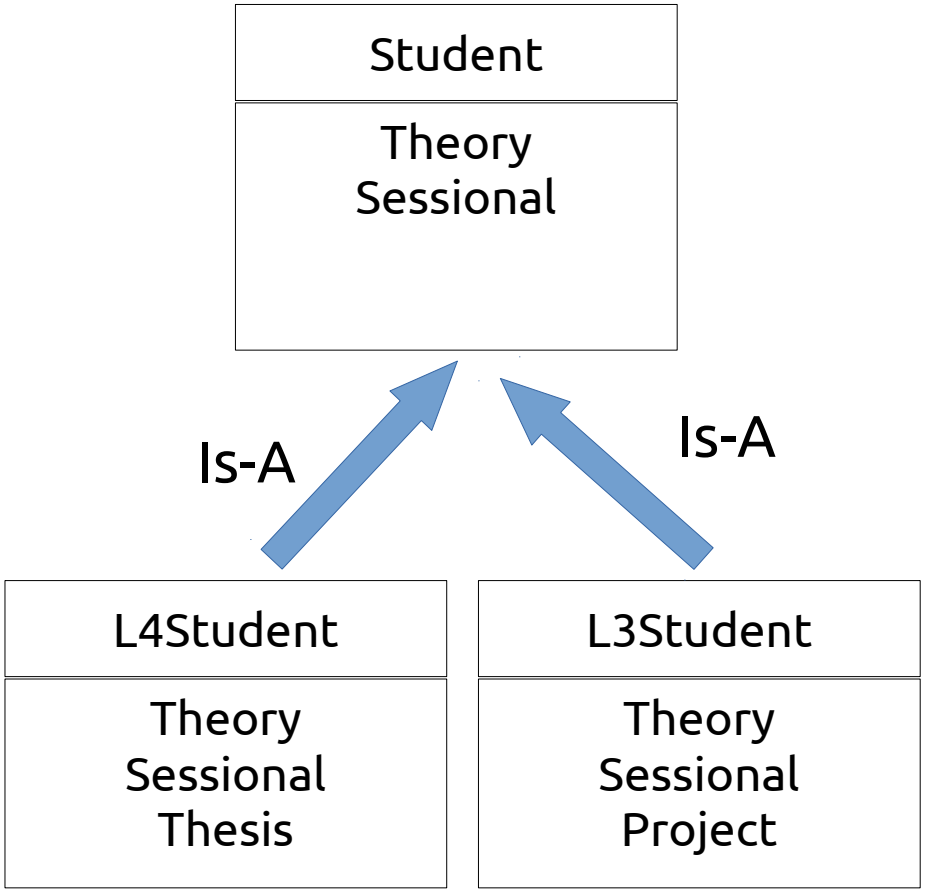
(1/3) Inheritance

Single Inheritance



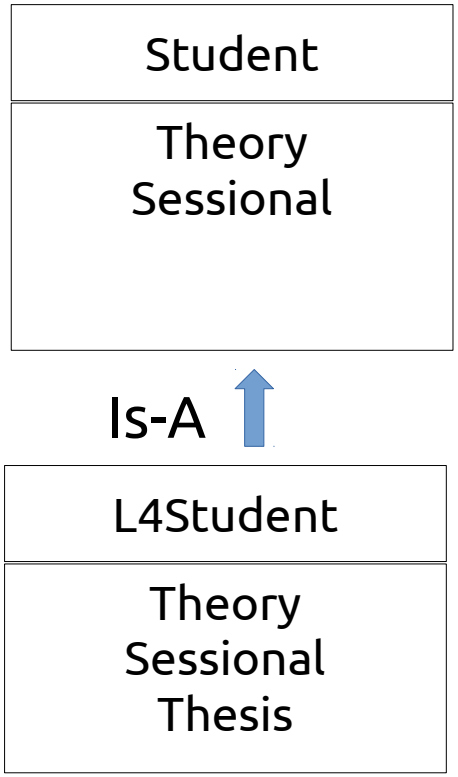
(1/3) Inheritance

Hierarchical Inheritance



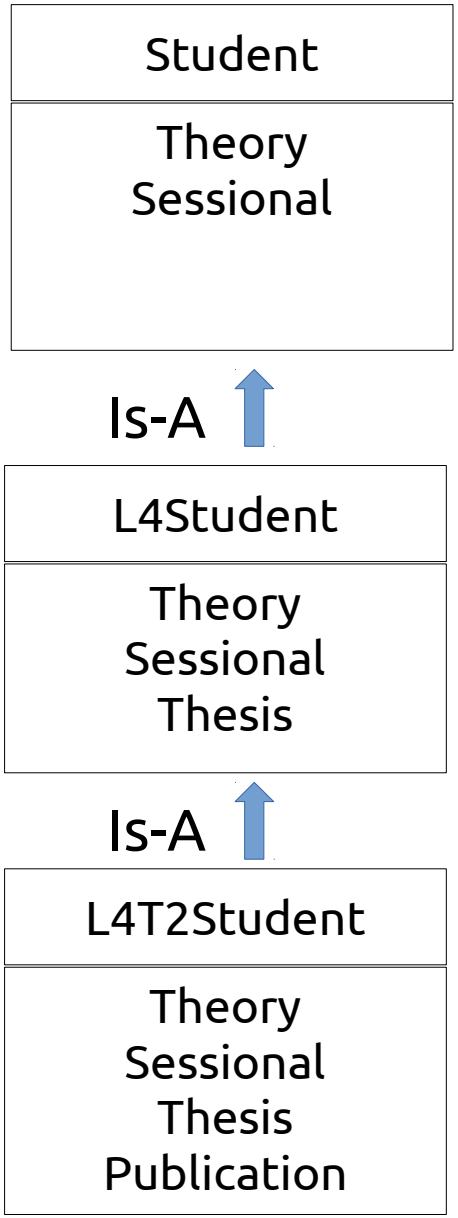
(1/3) Inheritance

Multi-level Inheritance



(1/3) Inheritance

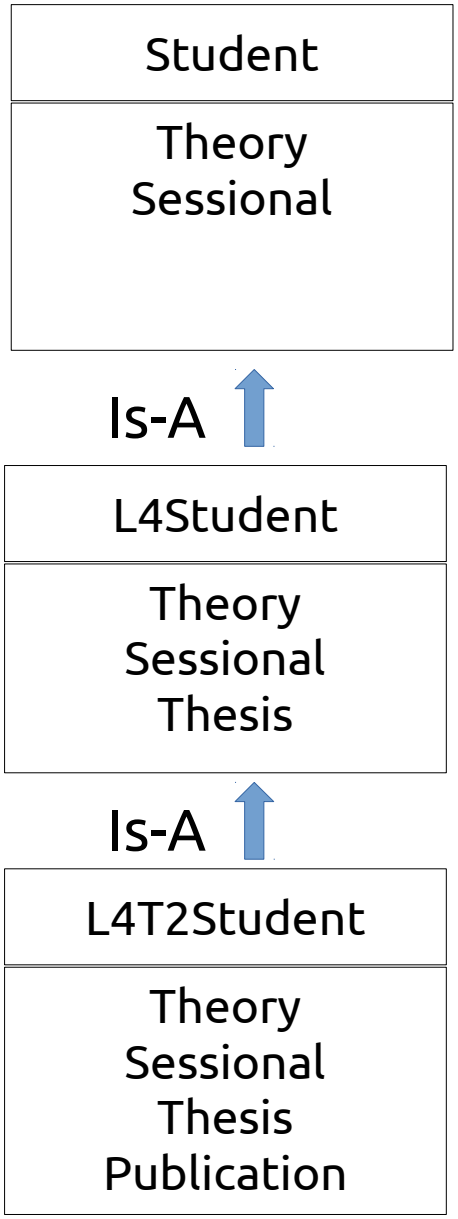
Multi-level Inheritance



(1/3) Inheritance

Multi-level Inheritance

```
class L4T2Student extends L4Student
{
    int publication;
    public L4T2Student(int theory,
            int sessional,
            int thesis,
            int publication)
    {
        super(theory, sessional, thesis);
        this.publication = publication;
    }
}
```

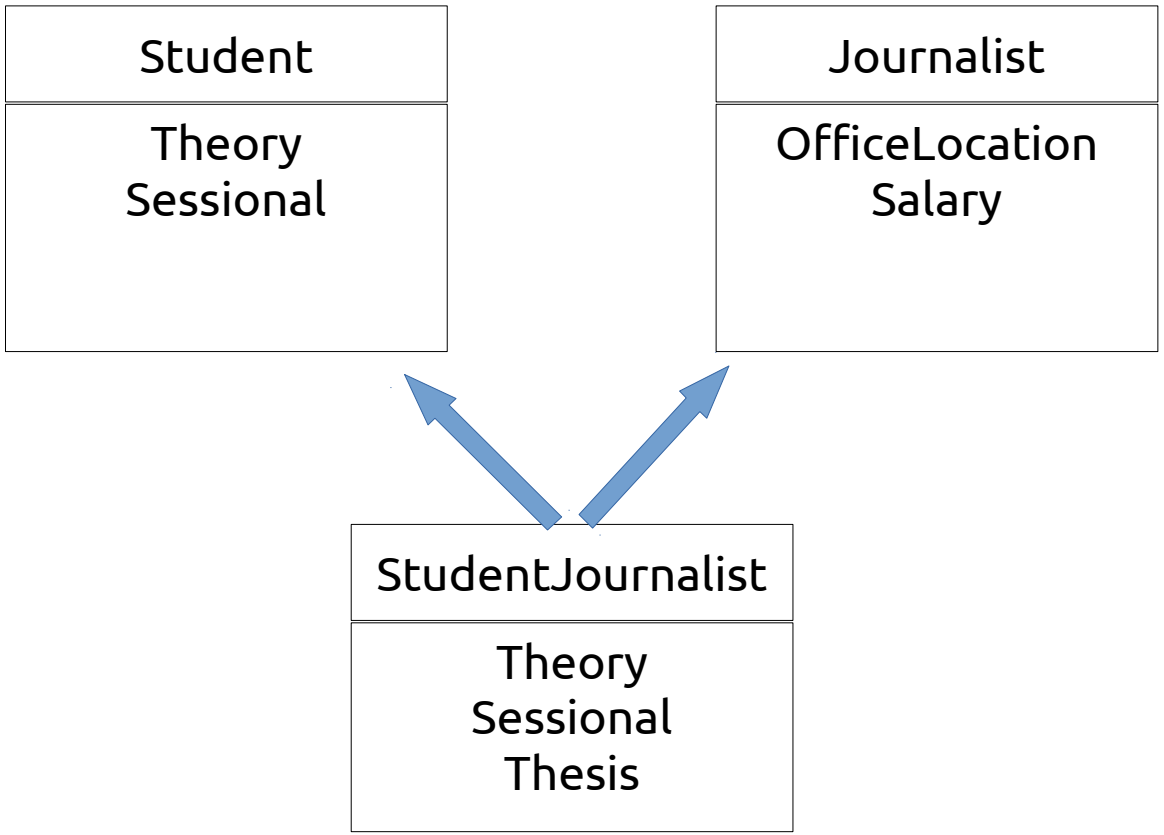


(1/3) Inheritance

Multiple Inheritance?

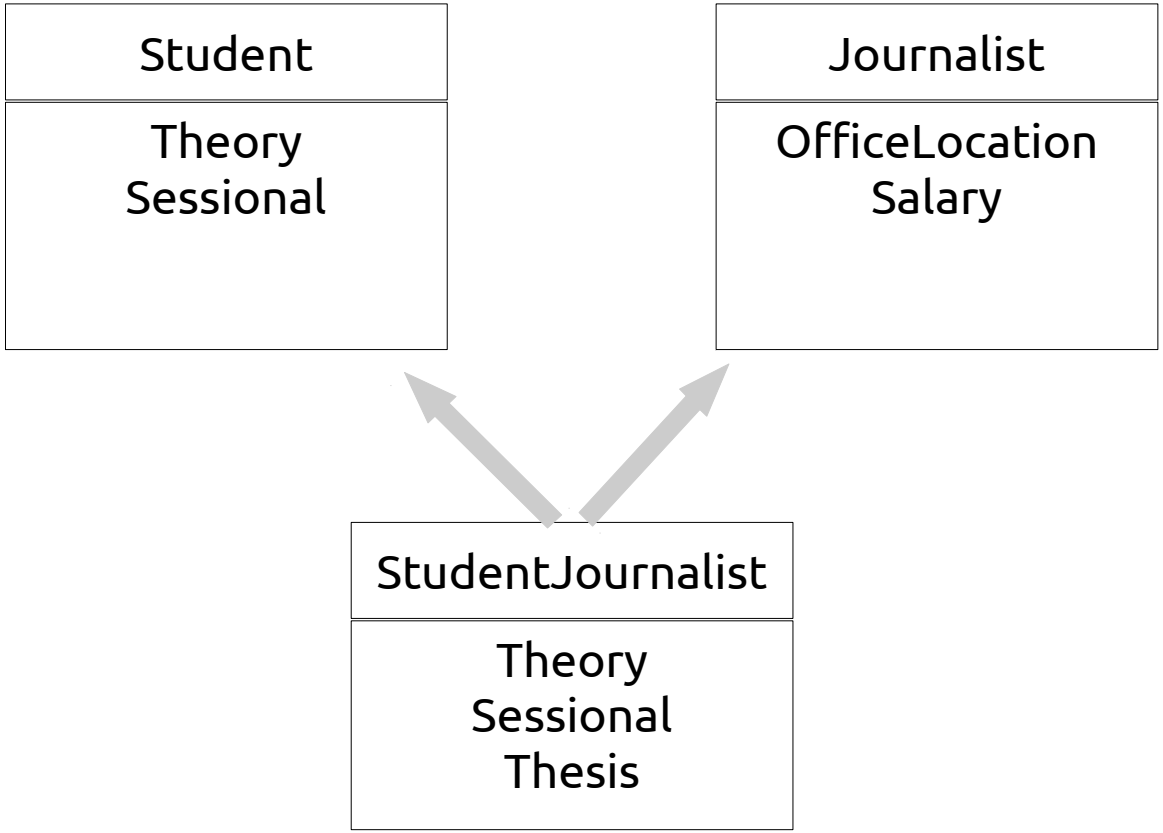
(1/3) Inheritance

Multiple Inheritance?



(1/3) Inheritance

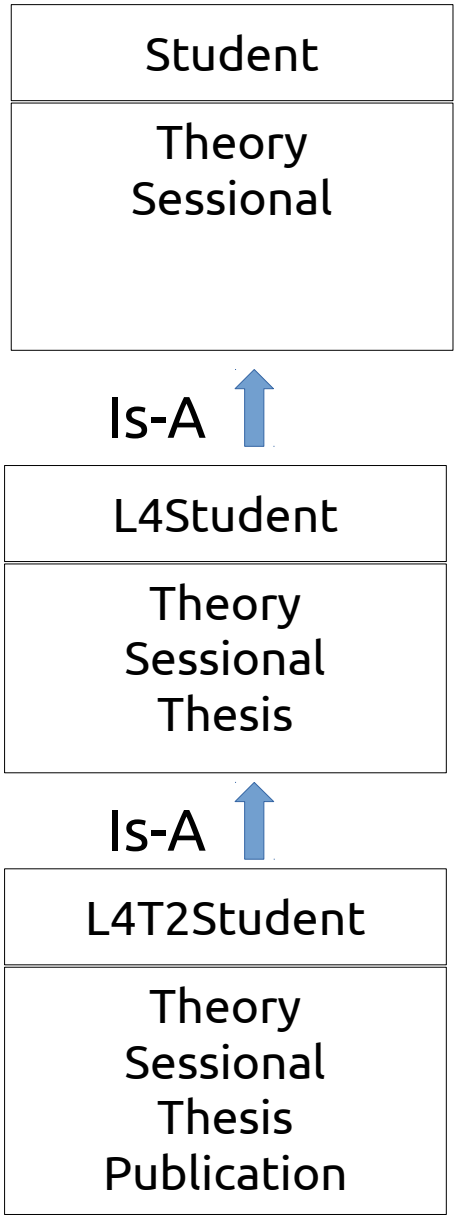
Multiple Inheritance? **Not allowed <u>in classes</u>.**



(1/3) Inheritance

Creating instance

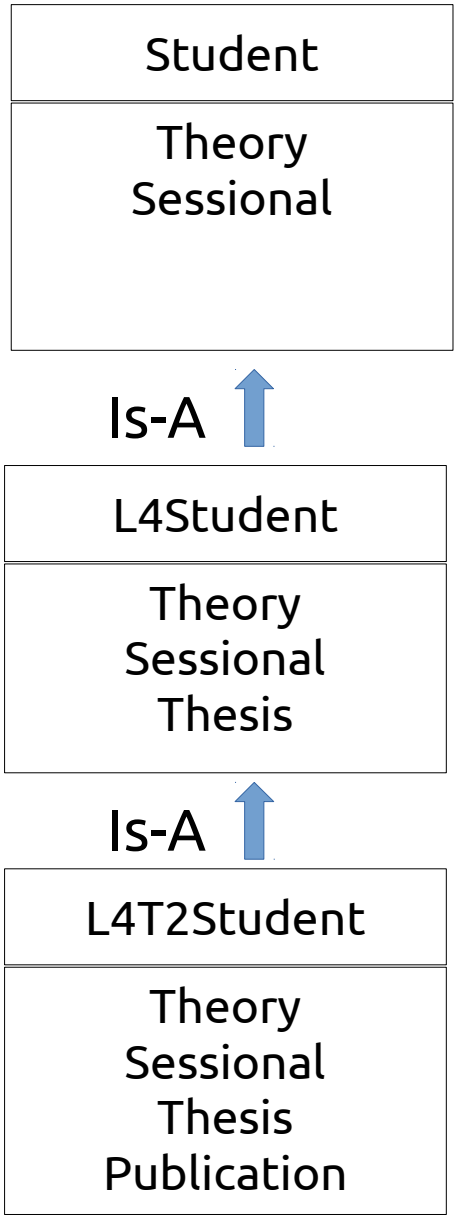
```
public class LabDemo {
    public static void main(String[] args) {
        Student s1 = new L4Student(10,10,10);
    }
}
```



(1/3) Inheritance

Implicit casting (Upcasting)

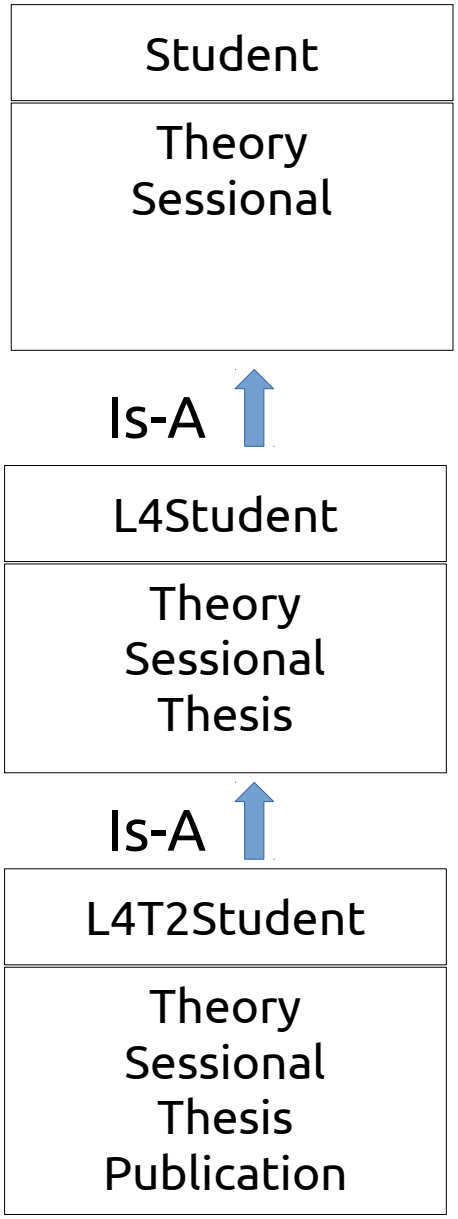
```
public class LabDemo {
    public static void main(String[] args) {
        Student s1 = new L4Student(10,10,10);
    }
}
```



(1/3) Inheritance

Explicit casting (Downcasting)

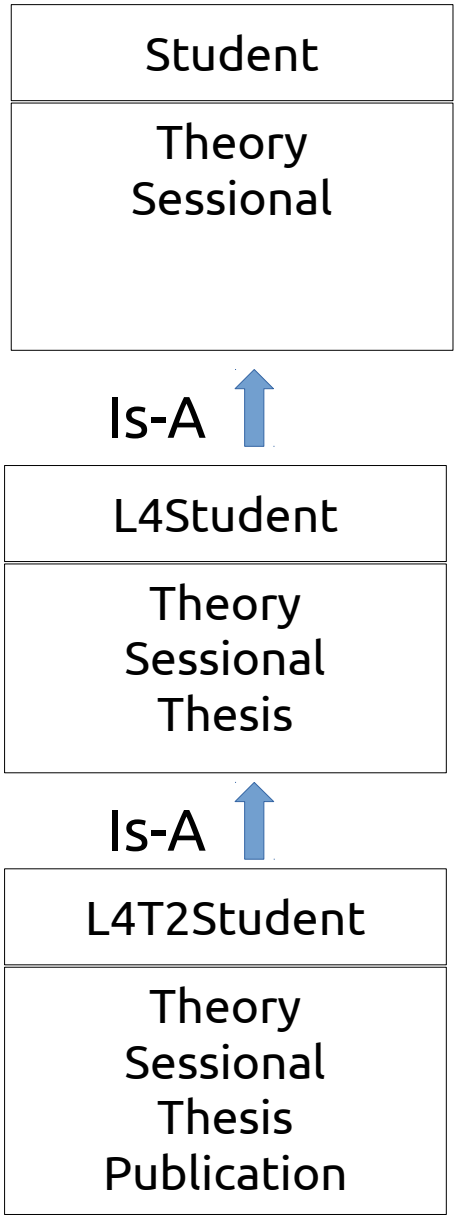
```
public class LabDemo {
    public static void main(String[] args) {
        L4T2Student s1 = new L4Student(10,10,10);
    }
}
```



(1/3) Inheritance

Explicit casting (Downcasting)

```
public class LabDemo {
    public static void main(String[] args) {
        L4T2Student s1
        = (L4T2Student) new L4Student(10,10,10);
    }
```

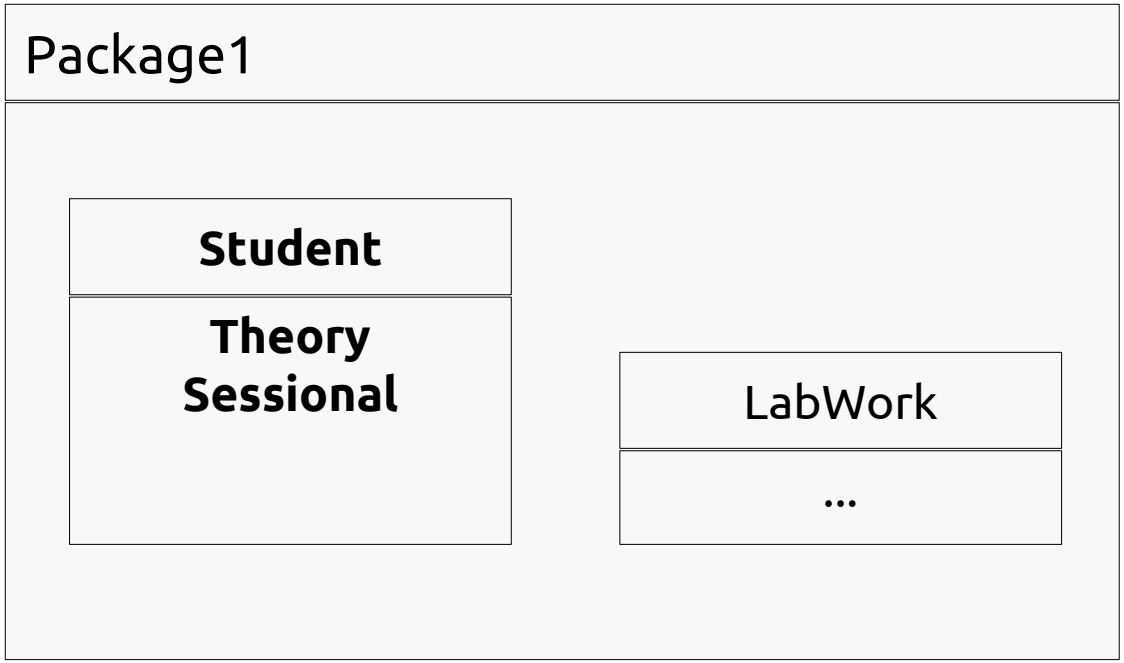


(2/3) Encapsulation

Hiding data and method from outside world

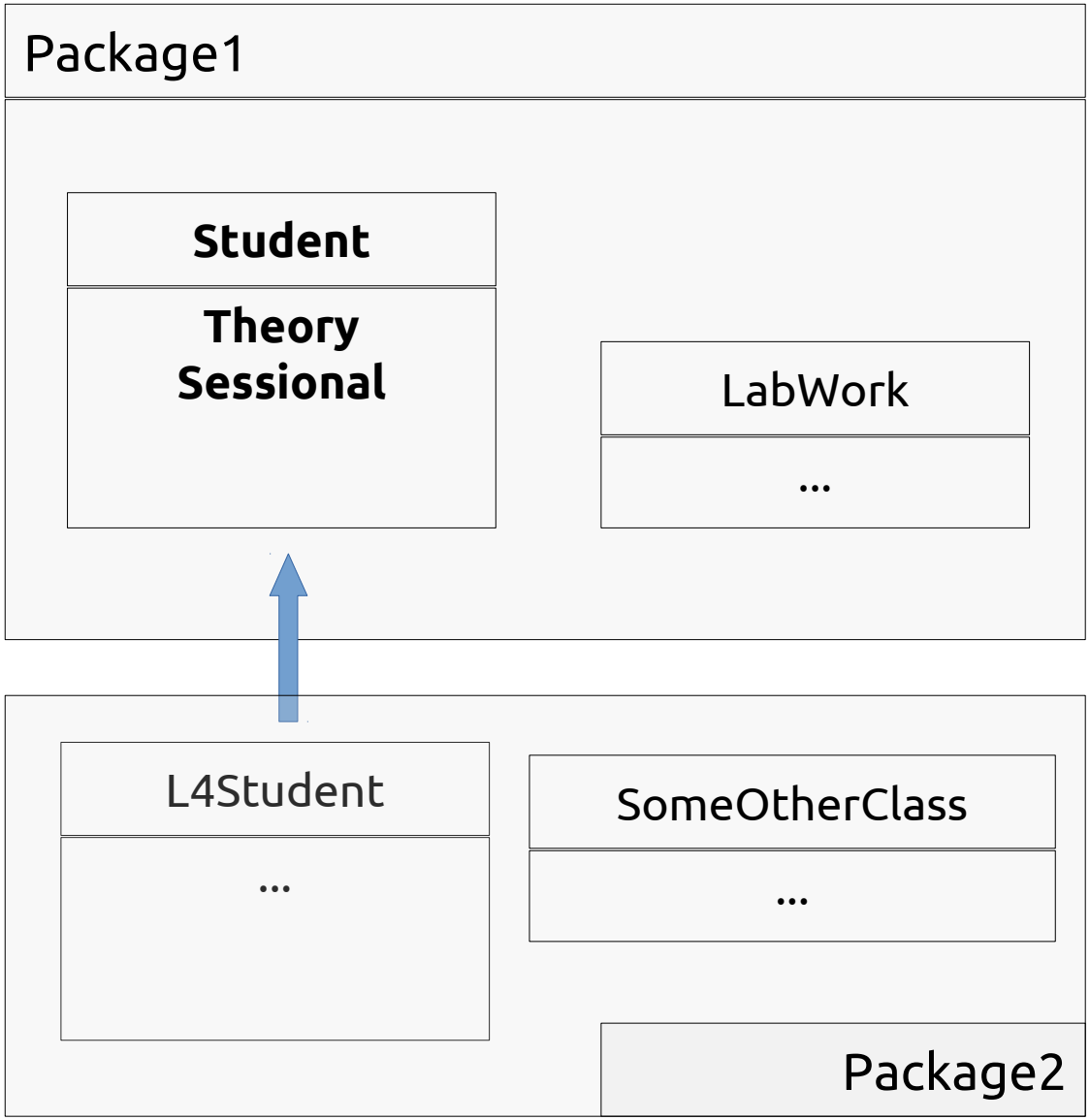
(2/3) Encapsulation

Hiding data and method from outside world



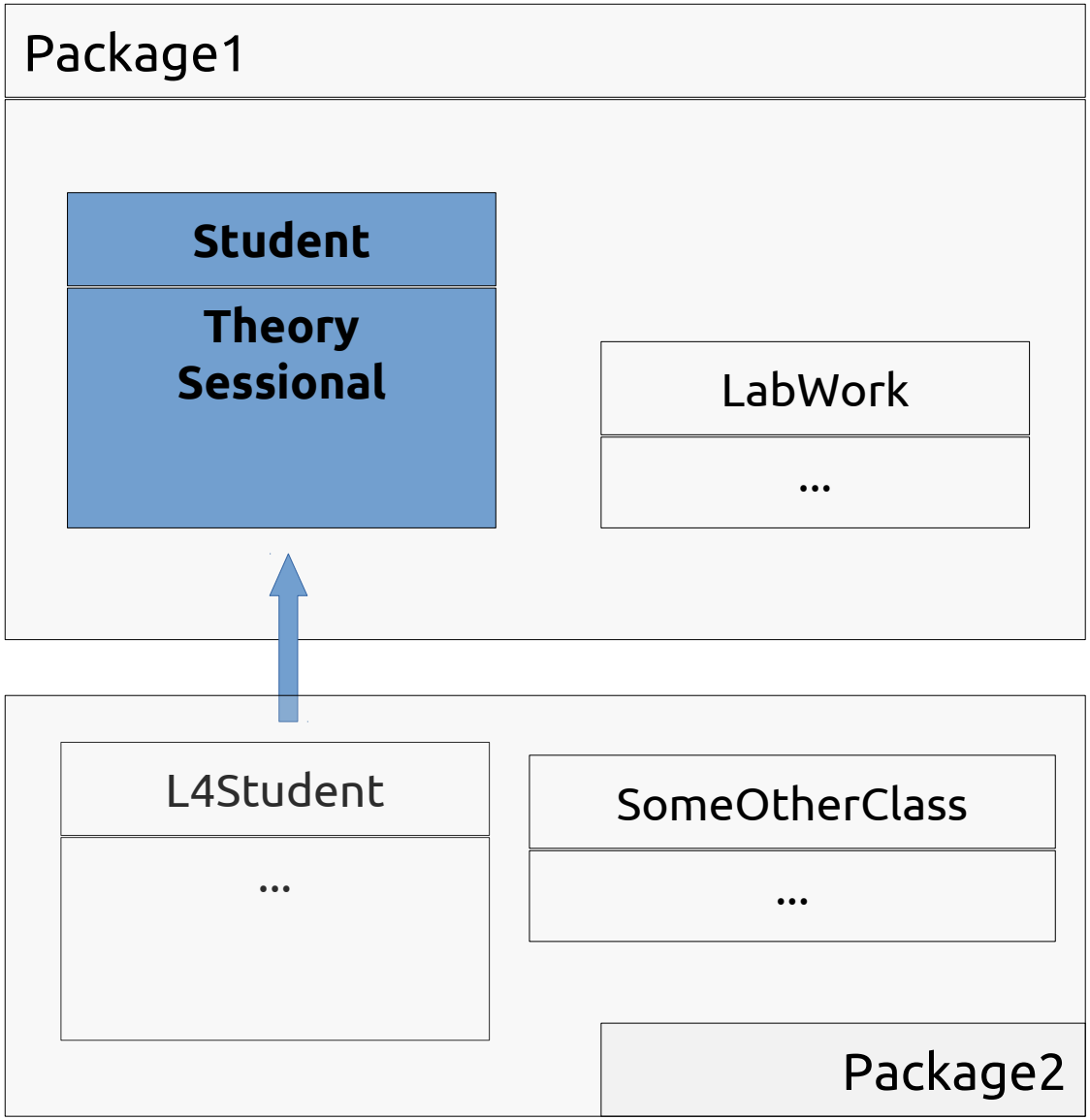
(2/3) Encapsulation

Hiding data and method from outside world



(2/3) Encapsulation

Hiding data and method from outside world

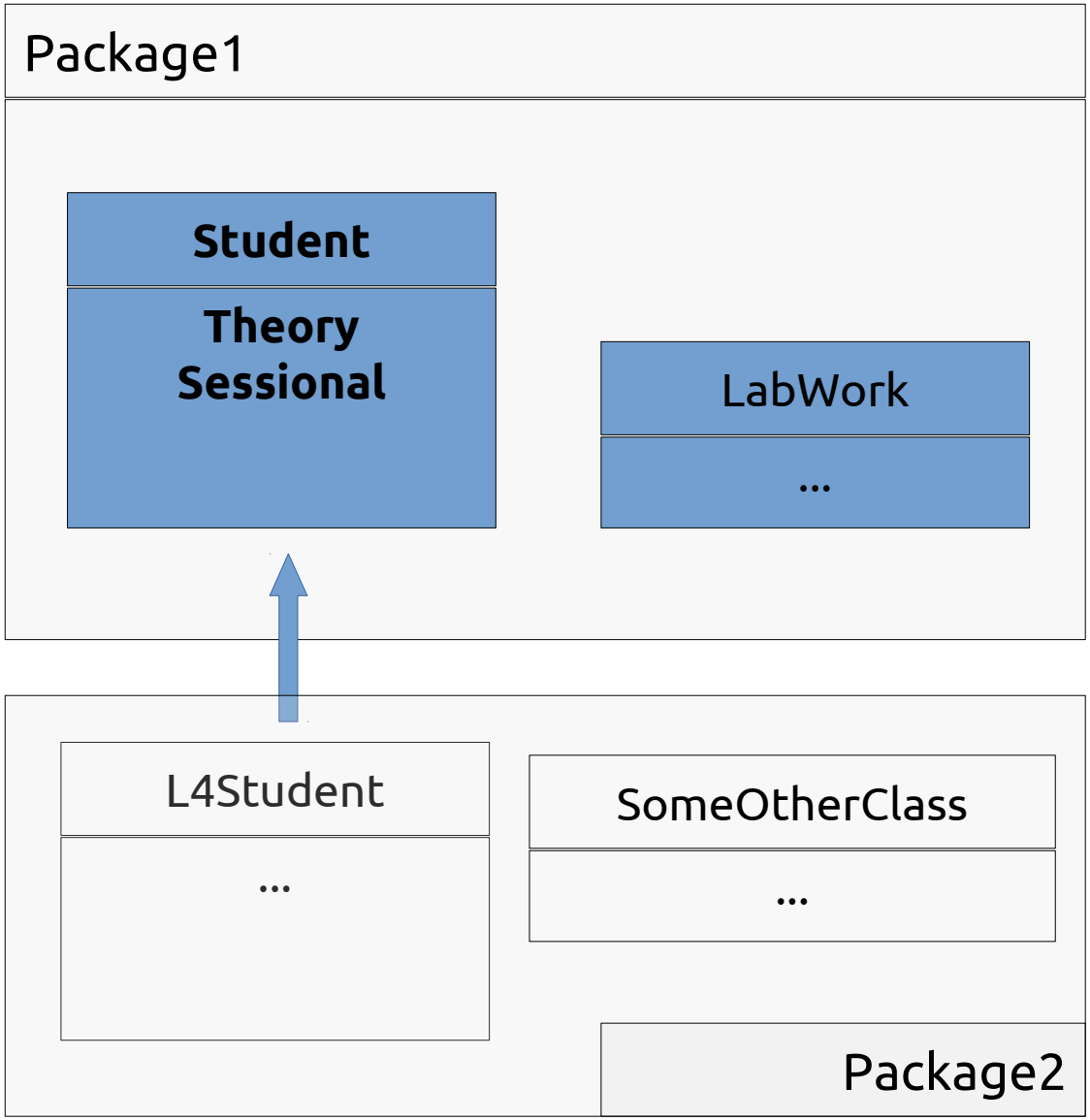


Access Modifier

Private

(2/3) Encapsulation

Hiding data and method from outside world

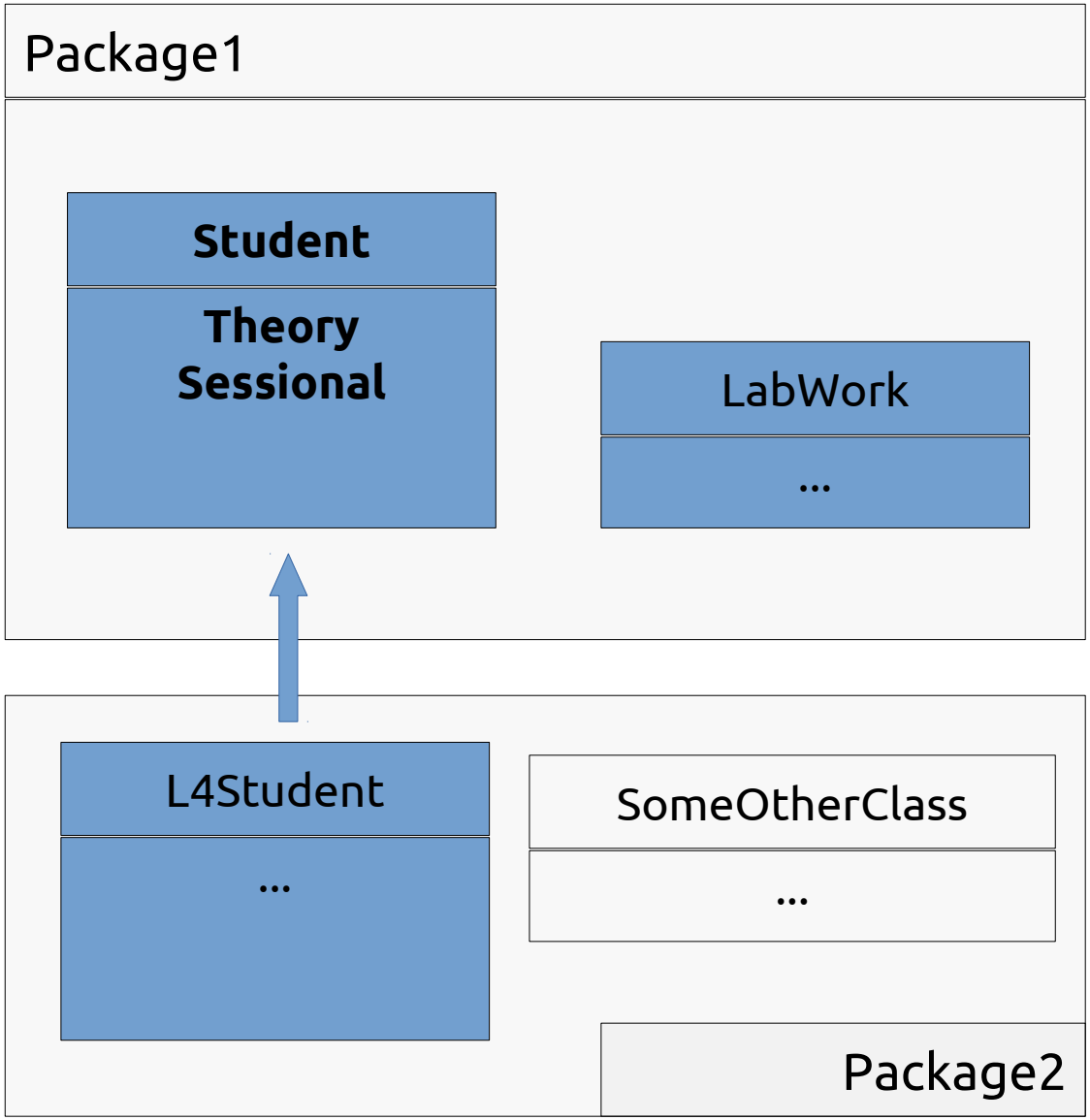


Access Modifier

No Modifier (Package)

(2/3) Encapsulation

Hiding data and method from outside world

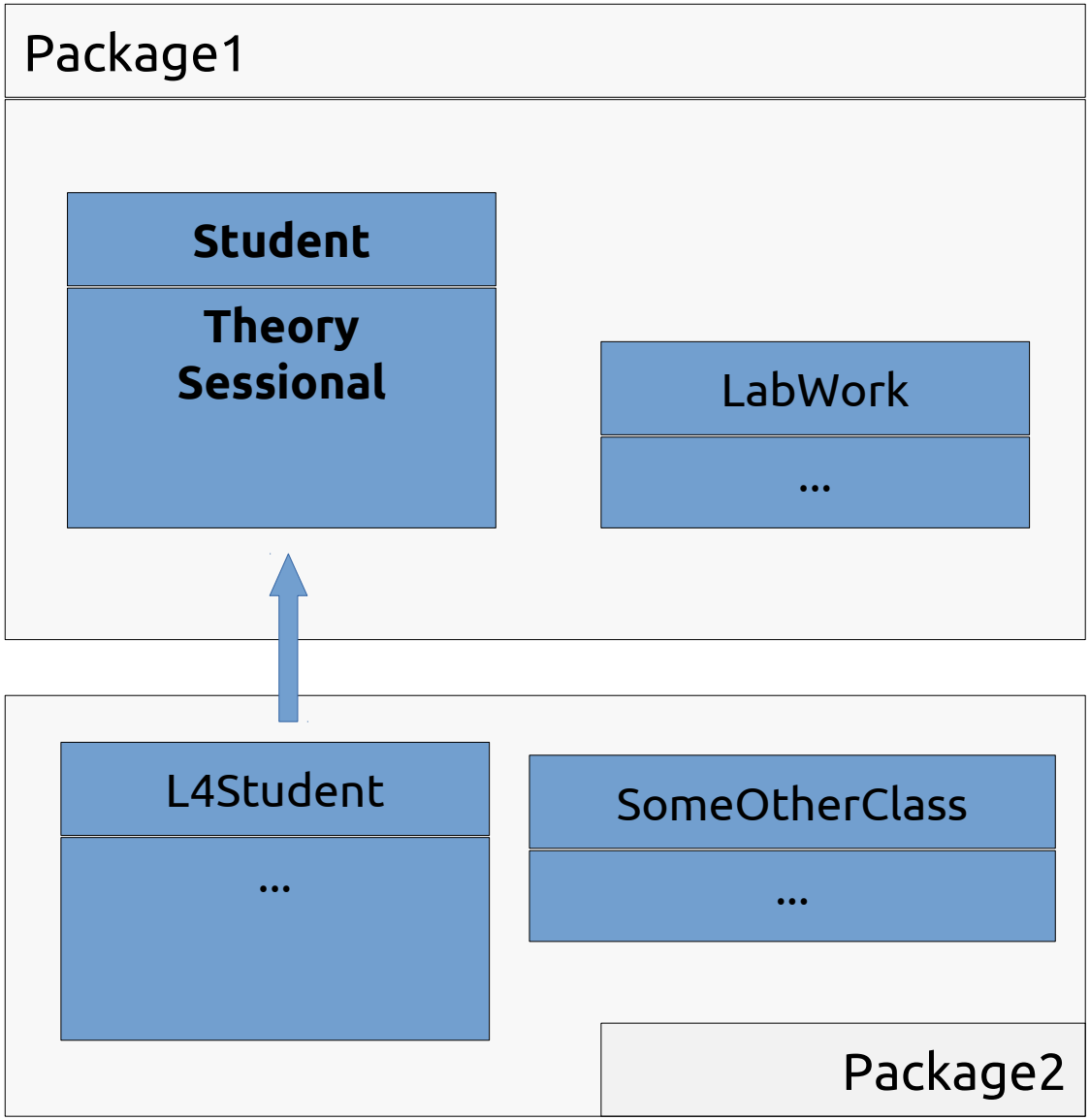


Access Modifier

Protected

(2/3) Encapsulation

Hiding data and method from outside world



Access Modifier

Public

(2/3) Encapsulation

Hiding data and method from outside world

Access Modifier

(2/3) Encapsulation

Hiding data and method from outside world

Access Modifier

Modifier	Class	Package	Subclass	World
`public`				
`protected`				
`no modifier*`				
`private`				

Image source: http://key-to-programming.blogspot.com/2015/09/java-modifiers.html

(2/3) Encapsulation

Hiding data and method from outside world

Access Modifier

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected				
*no modifier**				
private				

Image source: http://key-to-programming.blogspot.com/2015/09/java-modifiers.html

(2/3) Encapsulation

Hiding data and method from outside world

Access Modifier

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
*no modifier**				
private				

Image source: http://key-to-programming.blogspot.com/2015/09/java-modifiers.html

(2/3) Encapsulation

Hiding data and method from outside world

Access Modifier

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
*no modifier**	✓	✓	✗	✗
private				

Image source: http://key-to-programming.blogspot.com/2015/09/java-modifiers.html

(2/3) Encapsulation

Hiding data and method from outside world

Access Modifier

Modifier	Class	Package	Subclass	World
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
*no modifier**	✓	✓	✗	✗
private	✓	✗	✗	✗

Image source: http://key-to-programming.blogspot.com/2015/09/java-modifiers.html

(3/3) Polymorphism

One object/method taking multiple forms

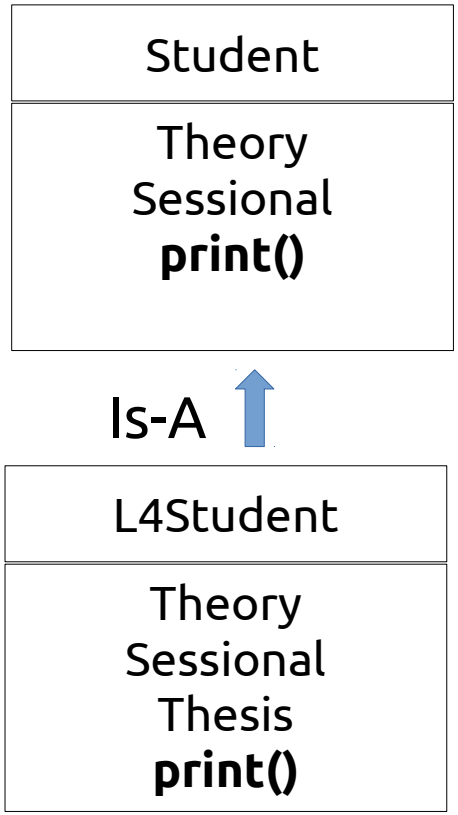
(3/3) Polymorphism

One object/method taking multiple forms

1. Method Overloading → Static Binding
2. Method Overriding → Dynamic Binding

(3/3) Polymorphism

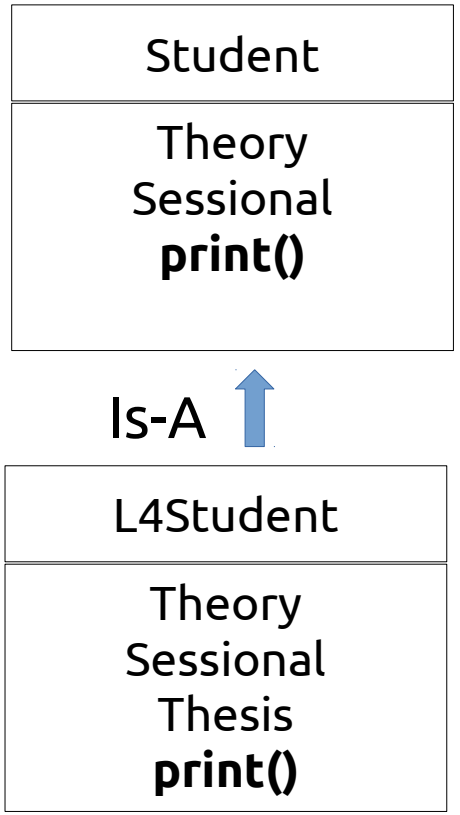
One object/method taking multiple forms



(3/3) Polymorphism

One object/method taking multiple forms

```
public static void main(String[] args) {
    Student s1 = new Student(10, 20);
    L4Student s2 = new L4Student(10, 20, 30);

    Student ref;

    ref = s1;
    ref.print();

    ref = s2;
    ref.print();
}
```

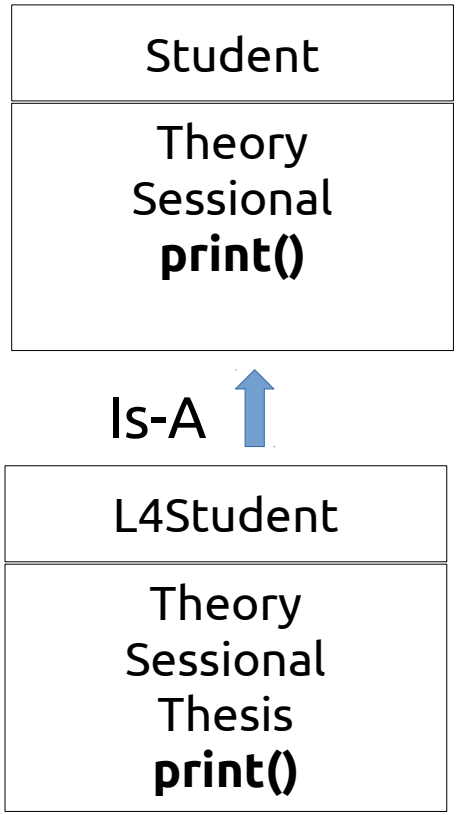


(3/3) Polymorphism

One object/method taking multiple forms

Dynamic Method Distpatch

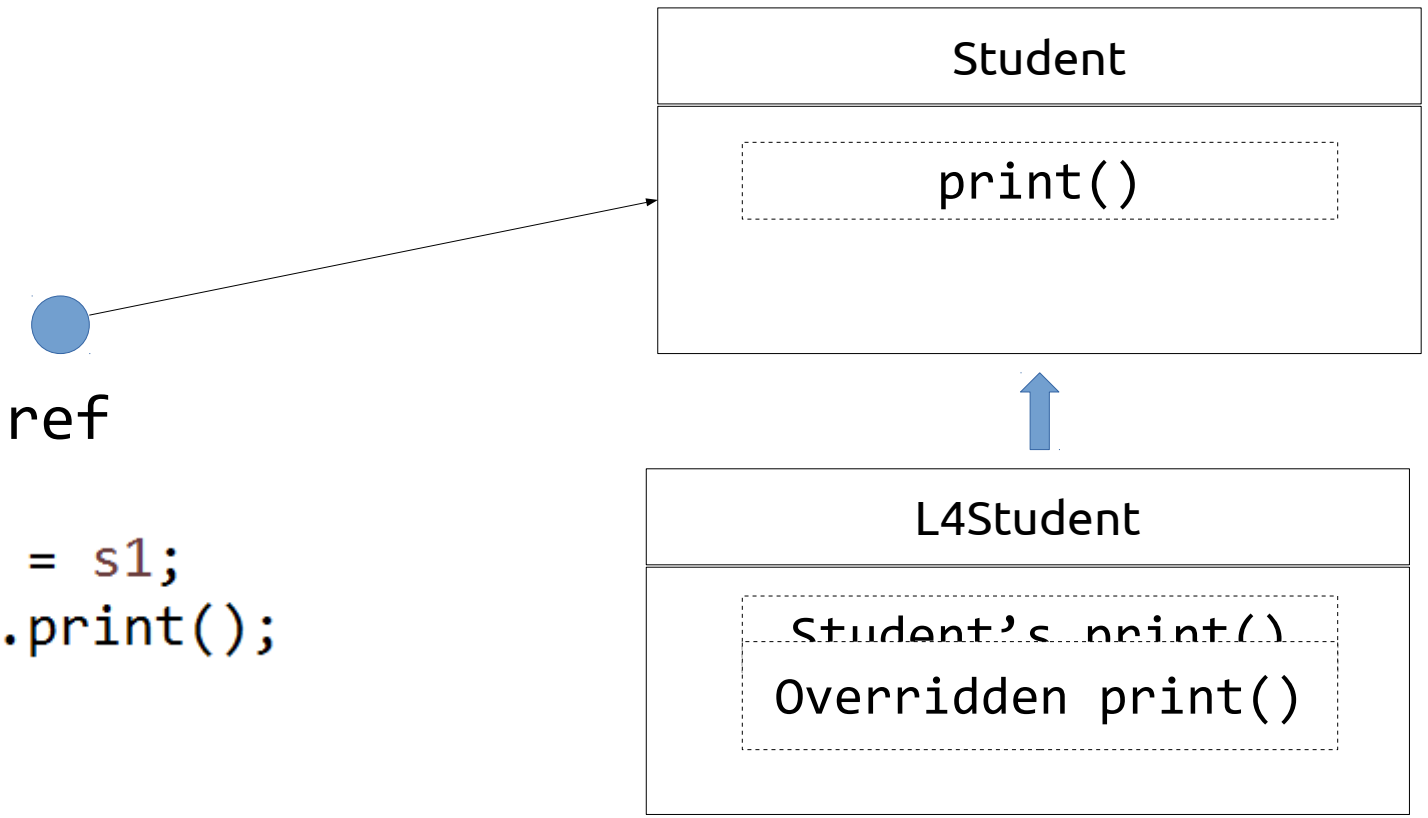
```
public static void main(String[] args) {
    Student s1 = new Student(10, 20);
    L4Student s2 = new L4Student(10, 20, 30);

    Student ref;

    ref = s1;
    ref.print();

    ref = s2;
    ref.print();
}
```



(3/3) Polymorphism

One object/method taking multiple forms

Dynamic Method Distpatch – What's happening?

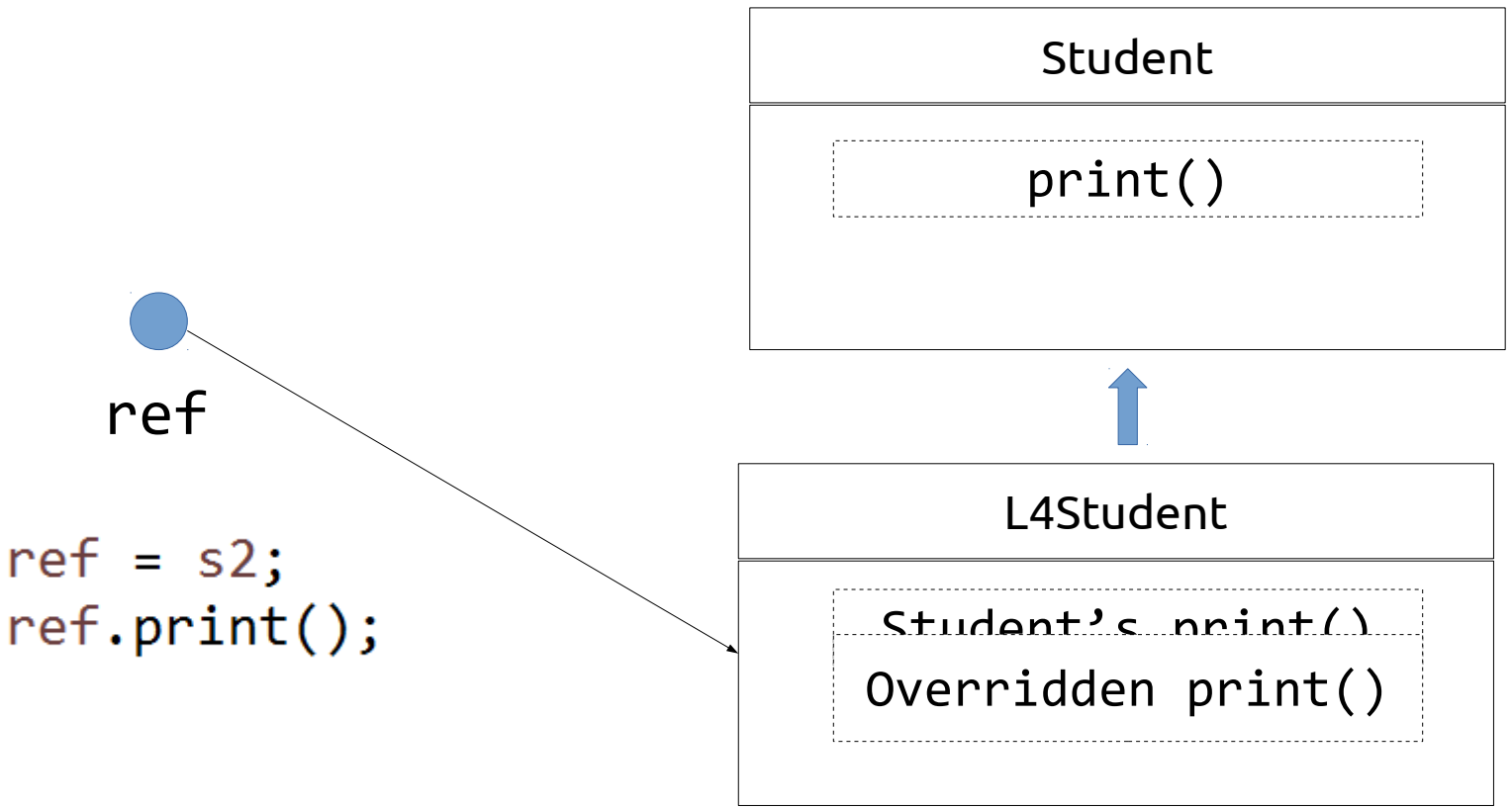



```
ref = s1;
ref.print();
```

(3/3) Polymorphism

One object/method taking multiple forms

Dynamic Method Distpatch – What's happening?




```
ref = s2;
ref.print();
```

(3/3) Polymorphism

One object/method taking multiple forms

Dynamic binding is resolved by looking at object, during runtime.

```
public static void main(String[] args) {
    Student s1 = new Student(10, 20);
    L4Student s2 = new L4Student(10, 20, 30);

    Student ref;

    ref = s1;
    ref.print();

    ref = s2;
    ref.print();
}
```

(3/3) Polymorphism

One object/method taking multiple forms

Static Binding

(3/3) Polymorphism

One object/method taking multiple forms

Static Binding

```
static void callPrinter(Student s)
{
    s.print();
}

static void callPrinter(L4Student s)
{
    s.print();
}
```

(3/3) Polymorphism

One object/method taking multiple forms

Static Binding

```
public static void main(String[] args) {
    Student s1 = new Student(10, 20);
    L4Student s2 = new L4Student(10, 20, 30);

    callPrinter(s1);
    callPrinter(s2);
}
```

Static Binding

```
static void callPrinter(Student s)
{
    s.print();
}

static void callPrinter(L4Student s)
{
    s.print();
}
```

(3/3) Polymorphism

One object/method taking multiple forms

Static Binding is resolved by looking at classtype, during compile time

```
public static void main(String[] args) {
    Student s1 = new Student(10, 20);
    L4Student s2 = new L4Student(10, 20, 30);

    callPrinter(s1);
    callPrinter(s2);
}
static void callPrinter(Student s)
{
    s.print();
}

static void callPrinter(L4Student s)
{
    s.print();
}
```

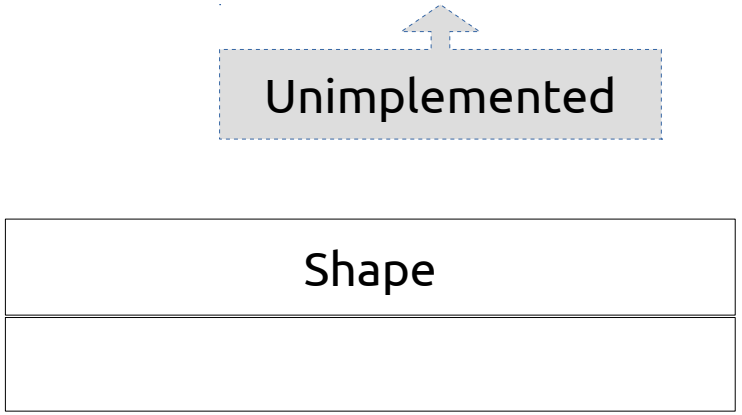
Abstract Class

A class that cannot be instantiated, & contains atleast one abstract method

Unimplemented

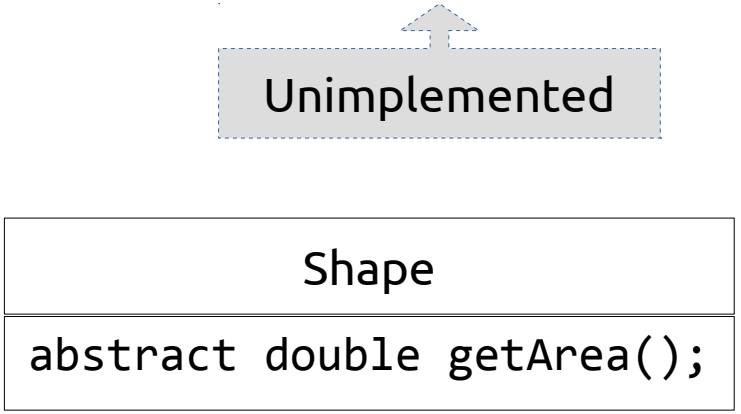
Abstract Class

A class that cannot be instantiated, & contains atleast one abstract method



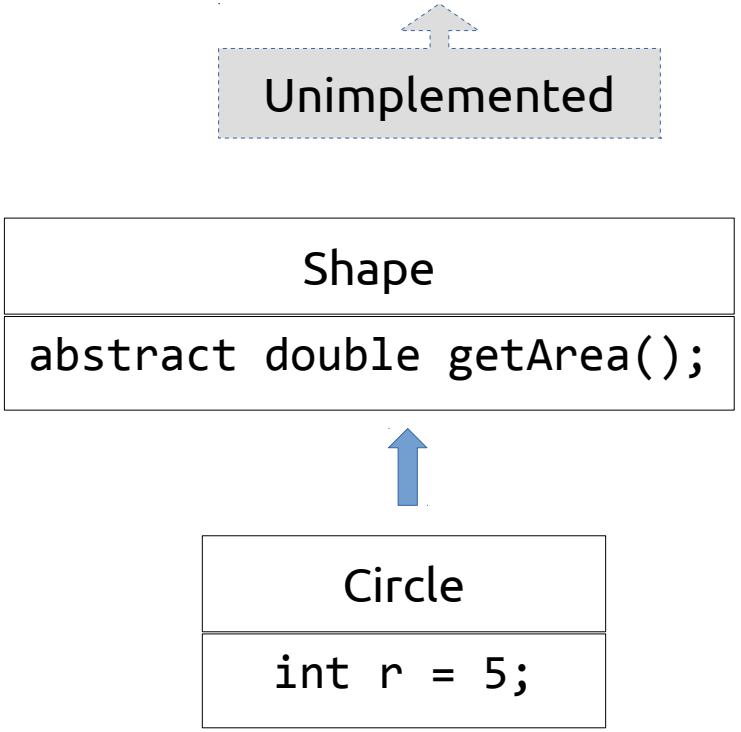
Abstract Class

A class that cannot be instantiated, & contains atleast one abstract method



Abstract Class

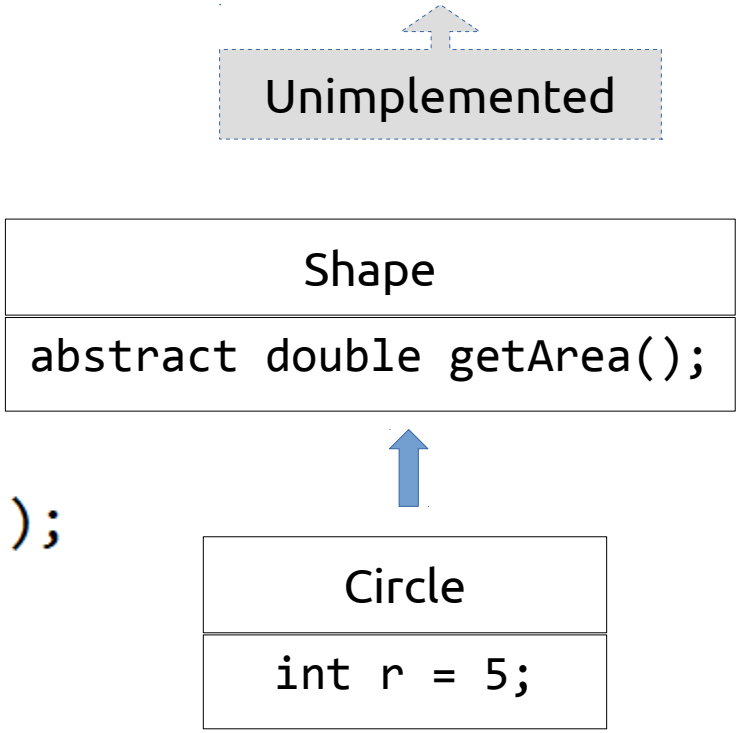
A class that cannot be instantiated, & contains atleast one abstract method



Abstract Class

A class that cannot be instantiated, & contains atleast one abstract method

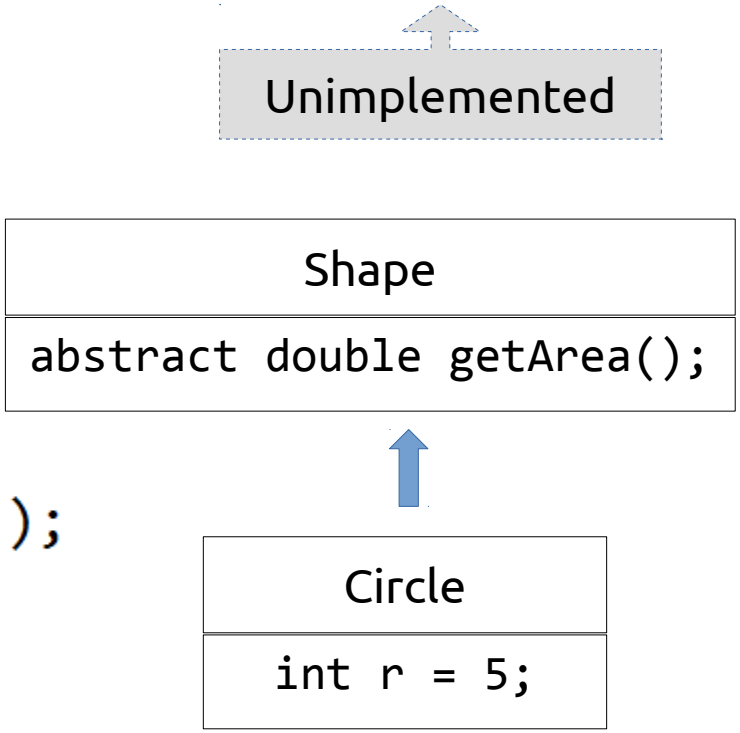
```
abstract class Shape
{
    void print()
    {
        System.out.println(
                "non-abstract method");
    }
    abstract double getArea();
}
```



Abstract Class

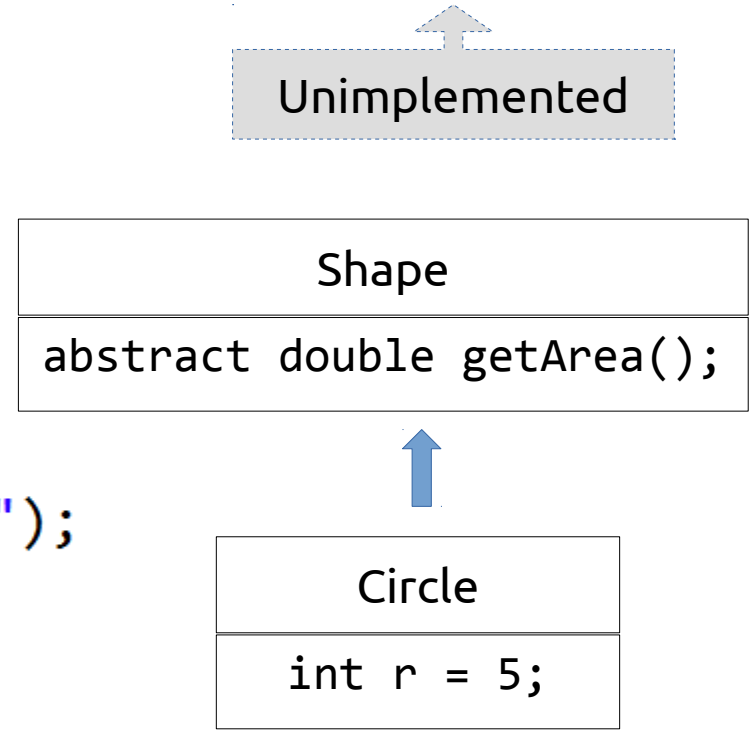
A class that cannot be instantiated, & contains atleast one abstract method

```
abstract class Shape
{
    void print()
    {
        System.out.println(
                "non-abstract method");
    }
    abstract double getArea();
}
class Circle extends Shape
{
    int r = 5;
    double getArea()
    {
        return 3.1416*r*r;
    }
}
```



Abstract Class

A class that cannot be instantiated, & contains atleast one abstract method




```
abstract class Shape
{
    void print()
    {
        System.out.println(
                "non-abstract method");
    }
    abstract double getArea();
}
class Circle extends Shape
{
    int r = 5;
    double getArea()
    {
        return 3.1416*r*r;
    }
}
```

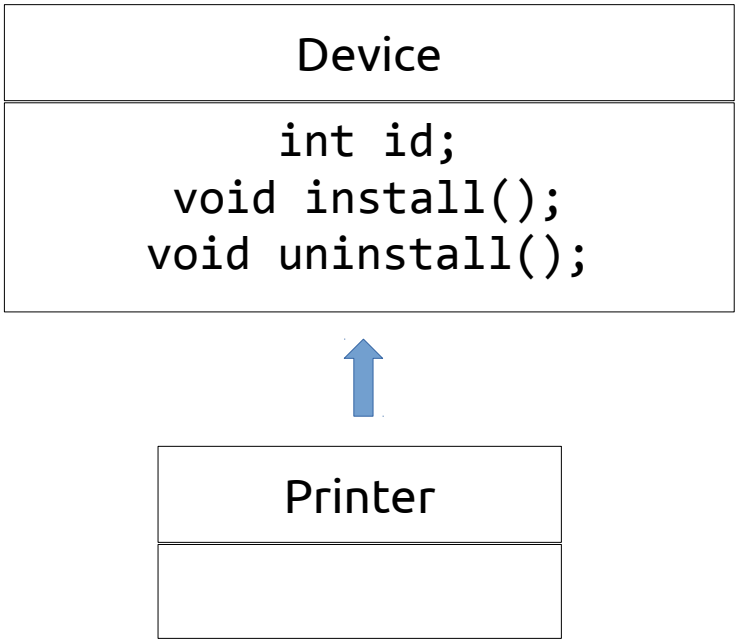
```
public static void main(String[] args) {
    Shape s1 = new Circle();
    System.out.println(s1.getArea());
}
```

Interface

A class that cannot be instantiated, & all it’s methods are abstract

Interface

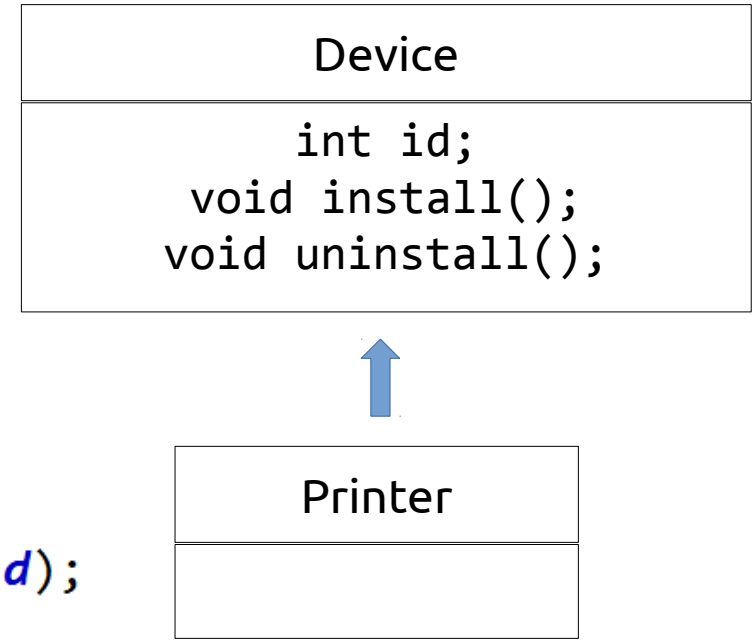
A class that cannot be instantiated, & all it’s methods are abstract



Interface

A class that cannot be instantiated, & all it's methods are abstract

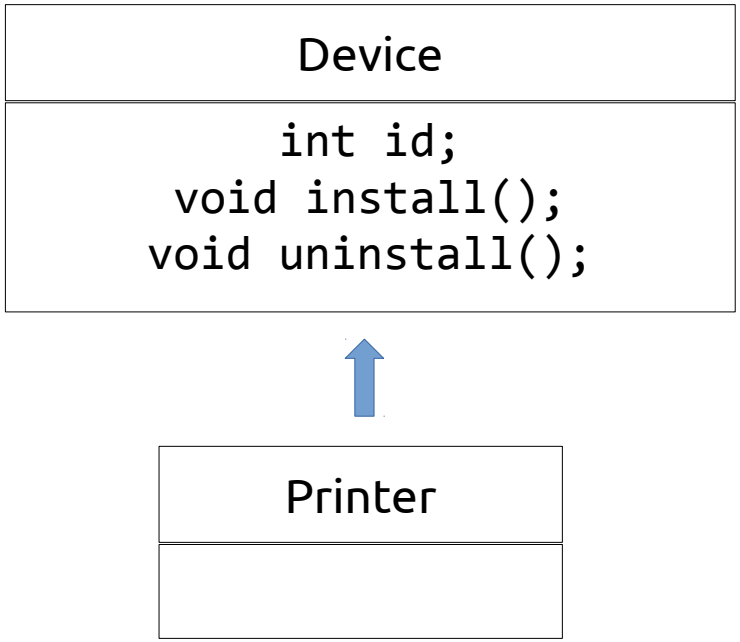
```
interface Device
{
    int id = 5; //static and final variable
    abstract void install();
    abstract void uninstall();
}

class Printer implements Device
{

    public void install()
    {
        System.out.println(
                "Installing Printer " + id);
    }
    public void uninstall()
    {
        System.out.println(
                "Uninstalling Printer " + id);
    }
}
```



Interface

A class that cannot be instantiated, & all it's methods are abstract

```
public static void main(String[] args) {
    System.out.println(Device.id);
    Device d1 = new Printer();
    d1.install();
    d1.uninstall();
}
```



Interface

A class that cannot be instantiated, & all it’s methods are abstract

Two things to note:

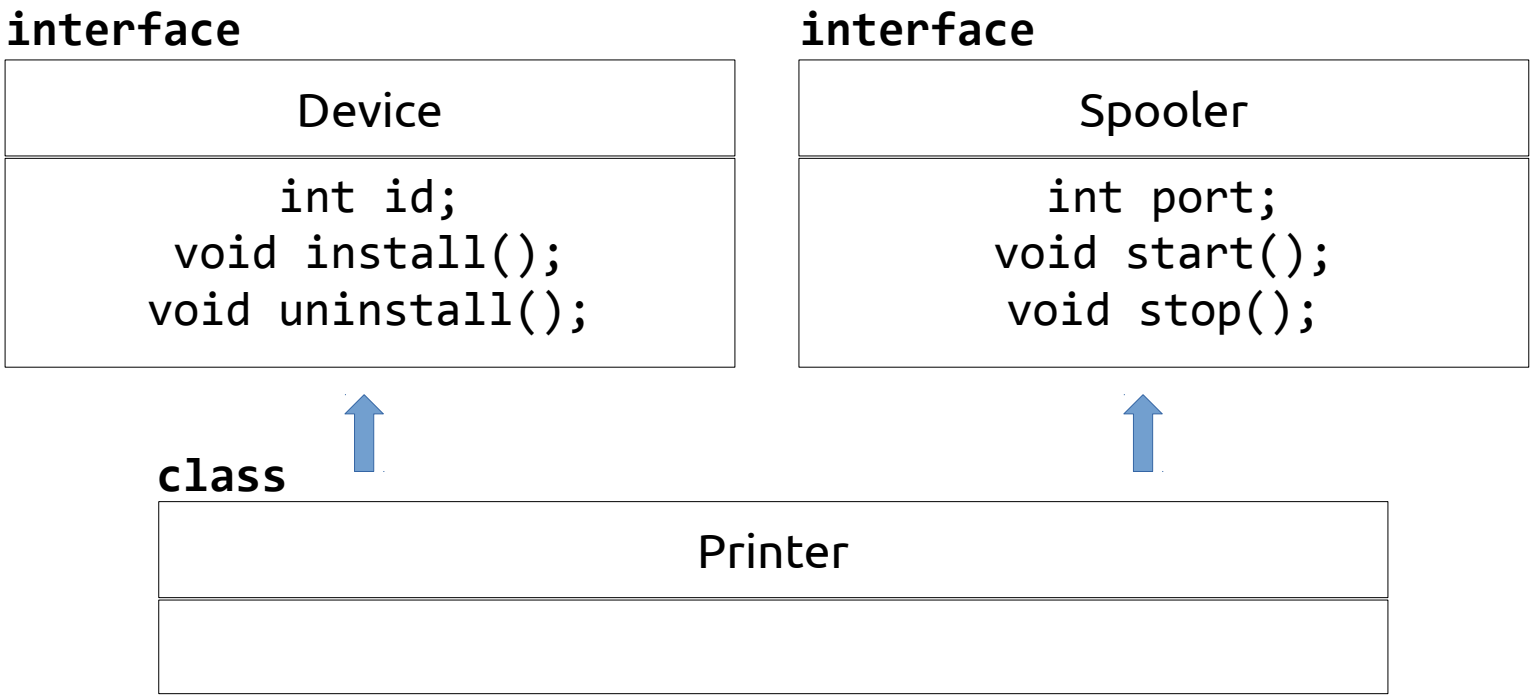
1. Interfaces can have **<u>multiple inheritance</u>.**

Interface

A class that cannot be instantiated, & all it's methods are abstract

Two things to note:

1. Interfaces can have **<u>multiple inheritance</u>.**

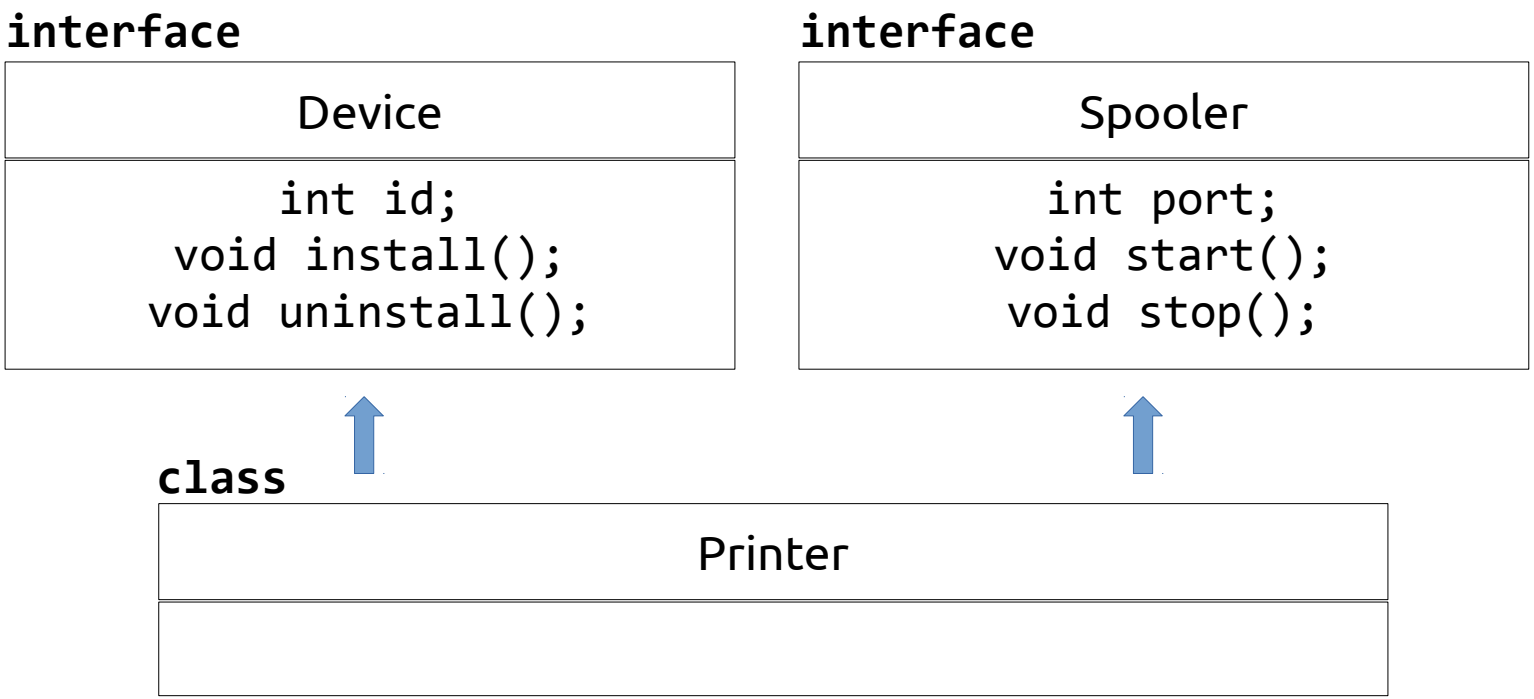


Interface

A class that cannot be instantiated, & all it's methods are abstract

Two things to note:

1. Interfaces can have **multiple inheritance**.




```
class Printer implements Device, Spooler
```

Interface

A class that cannot be instantiated, & all it’s methods are abstract

Two things to note:

2. Interfaces be extended into another interface

```
interface PoweredDevice extends Device
{
    int power_consumption = 20; //kW
}
```

```
class Printer implements PoweredDevice, Spooler
```

Interface

A class that cannot be instantiated, & all it's methods are abstract

Two things to note:

2. Take a look at this diagram below

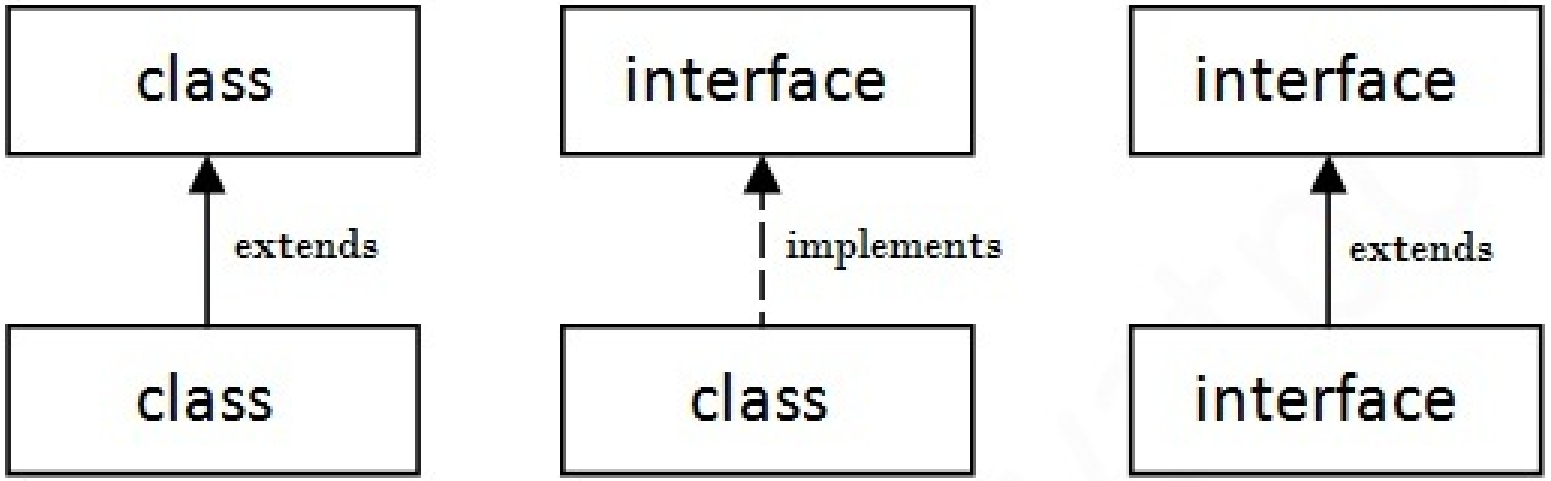


Image Source: https://www.javatpoint.com/interface-in-java

Interface

A class that cannot be instantiated, & all it's methods are abstract

Two things to note:

2. It means we can extend an interface into abstract class too.

```
abstract class SolarPoweredDevice implements Device
{
    public void install()
    {
        System.out.println(
                "Installing Solar Support");
    }
}
```