3 Fundamentals of OOP in Java

CSE-220,

Tasmiah Tamzid Anannya
Lecturer
Dept of CSE, MIST

Java Program

- The main building block of creating a Java Program is *class*.
- A *class* is a named block ({ }) that follows a keyword “class”.

Example:

```
class Hello{

}
```

- Here, “Hello” is the class name which follows the keyword “class”.
- A Java class may contain ***variable*** and ***function*** declarations. Generally, a function is called **method** in java.
- The file that contains the java codes known as **Java Source file** and each source file name has the extension *.java*.
- A Java source file can contain *one* or *more* class declaration.
- If a java source file contains only one class declaration, the file name must be given same as the class name in that file.

Java Programs (cont.)

- If a source file contains more than one java class declaration and **no class is declared as** *public*, the file name can be given according to any of the class name.
- If **a class is declared** *public*, the file name containing the class must be the same as the class name.
- The above rule implies that a source file can only contain at the most one public class.

Hello.java or Bye.java

```
class Hello{

}
class Bye{

}
```

Hello.java

```
public class Hello{

}
class Bye{

}
```

This file cannot be given Bye.java name

Java Programs (cont.)

•In order to create an application in Java, the program must have a class that defines a method or function named *main*, which is the starting point for the execution of any application.

•The main method is as follows:

```
public static void main( String[] args){
                    //some code here
        }
```

•Without main method, a program **can be compiled** but **can not be run**.

`public static void main( String[] args)`

- **public:**
  - This is *Access modifier* : specifies from where and who can access the method.
  - The *main()* method must be public so that JVM can invoke it from outside the class.

- **static**
  - When java runtime starts, there is no object of the class present. That's why the main method has to be static so that JVM can load the class into memory and call the main method.
  - If the main method won't be static, JVM would not be able to call it because there is no object of the class is present.
- void
  - Java programming mandates that every method provide the return type.
  - Java main method doesn't return anything, that's why it's return type is void.
  - Once the main method is finished executing, java program terminates. So there is no point in returning anything, there is nothing that can be done for the returned object by JVM.

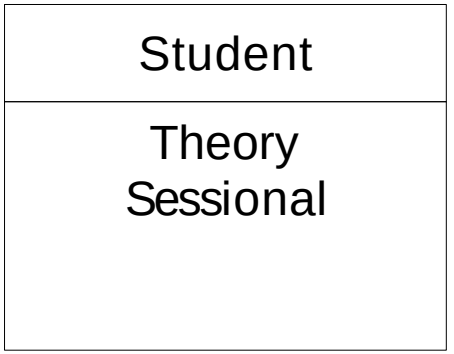
`public static void main( String[] args)`

- **main**
  - This is the name of java main method.
  - It's fixed and when we start a java program, it looks for the main method.

- **string[] args**
  - Java main method accepts a single argument of type String array.
  - This is also called as java command line arguments.
  - You can change the name of String array argument. For example you can change ***args*** to ***myStringArgs***.

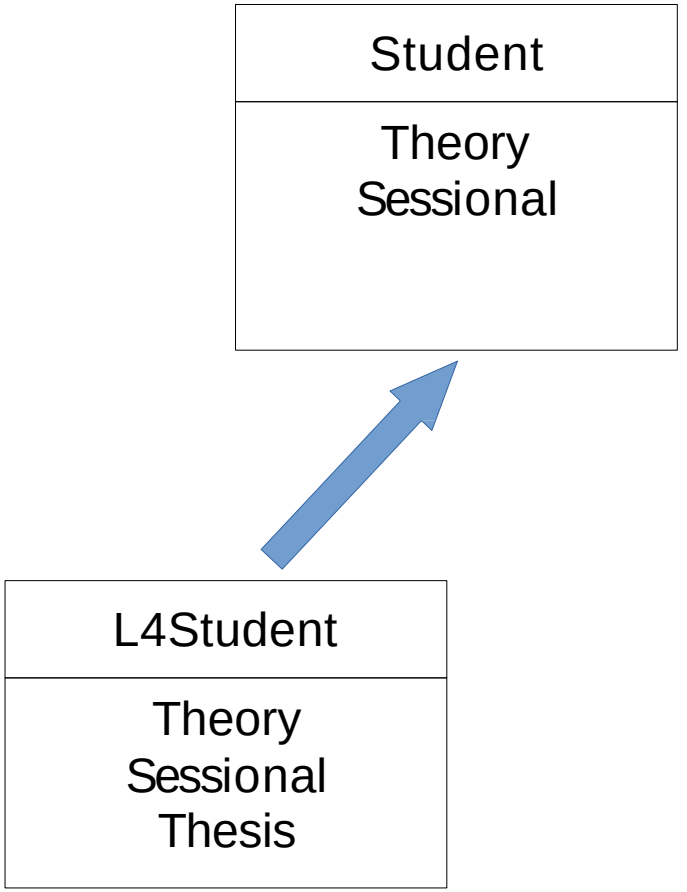
(1/3) Inheritance

Going from general to specific



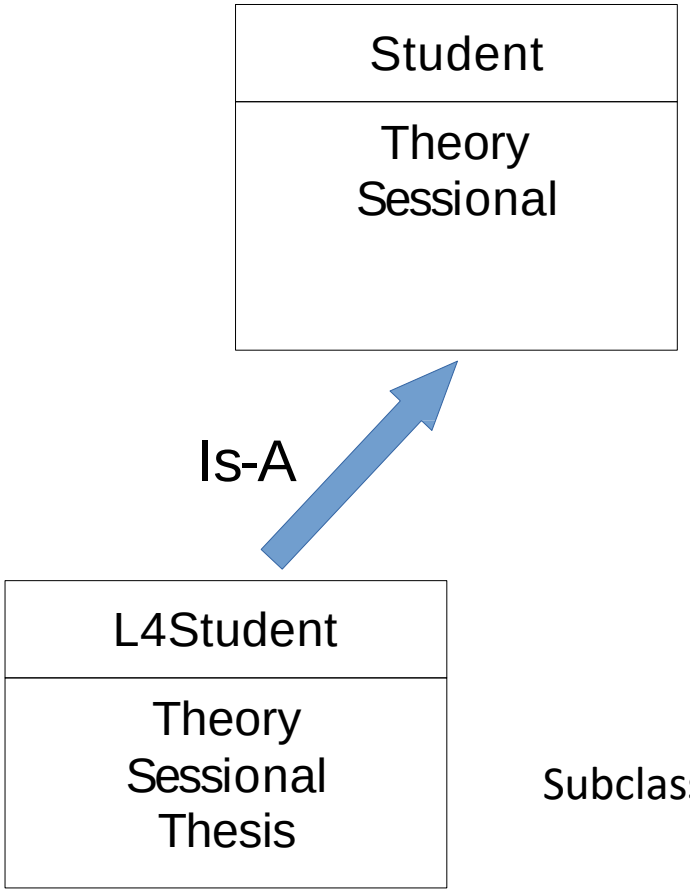
(1/3) Inheritance

Going from general to specific



(1/3) Inheritance

Going from general to specific



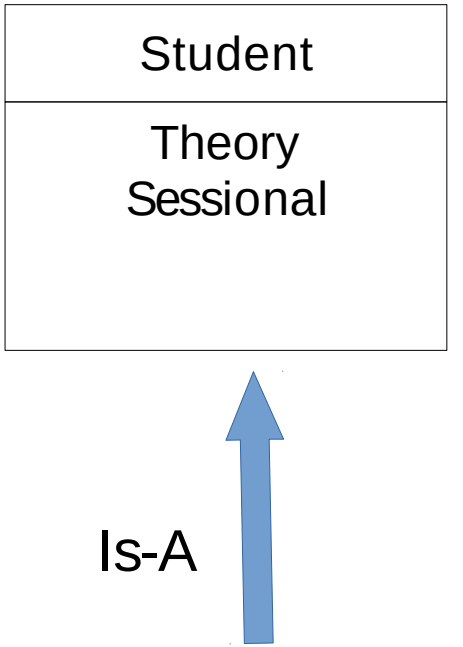
Superclass/base class/parent class

Subclass/derived class/child class

(1/3) Inheritance

Constructor

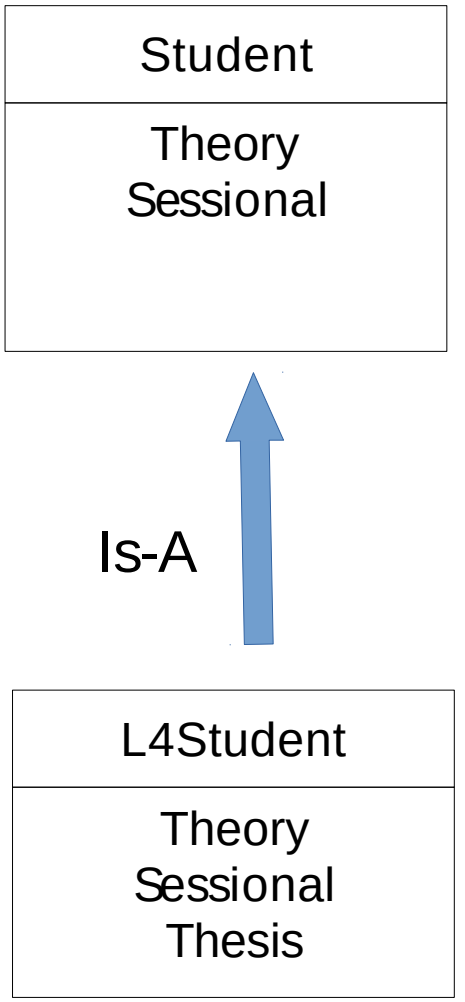
```
class Student
{
    int theory, sessional;
    public Student(int theory, int sessional)
    {
        this.theory = theory;
        this.sessional = sessional;
    }
}
class L4Student extends Student
{
    int thesis;









}
```



(1/3) Inheritance

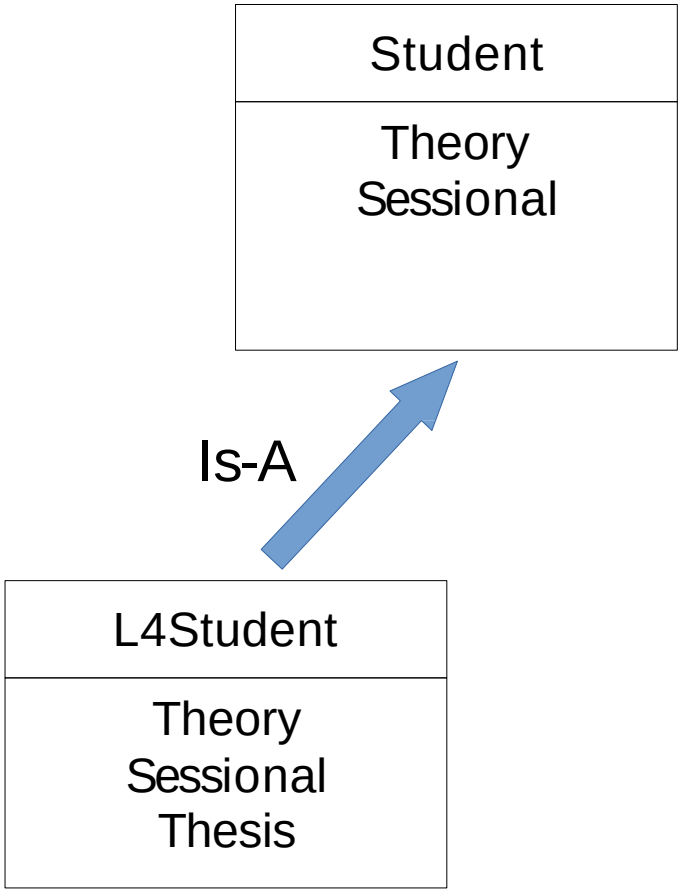
Super Constructor

```
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;
    }
}
```



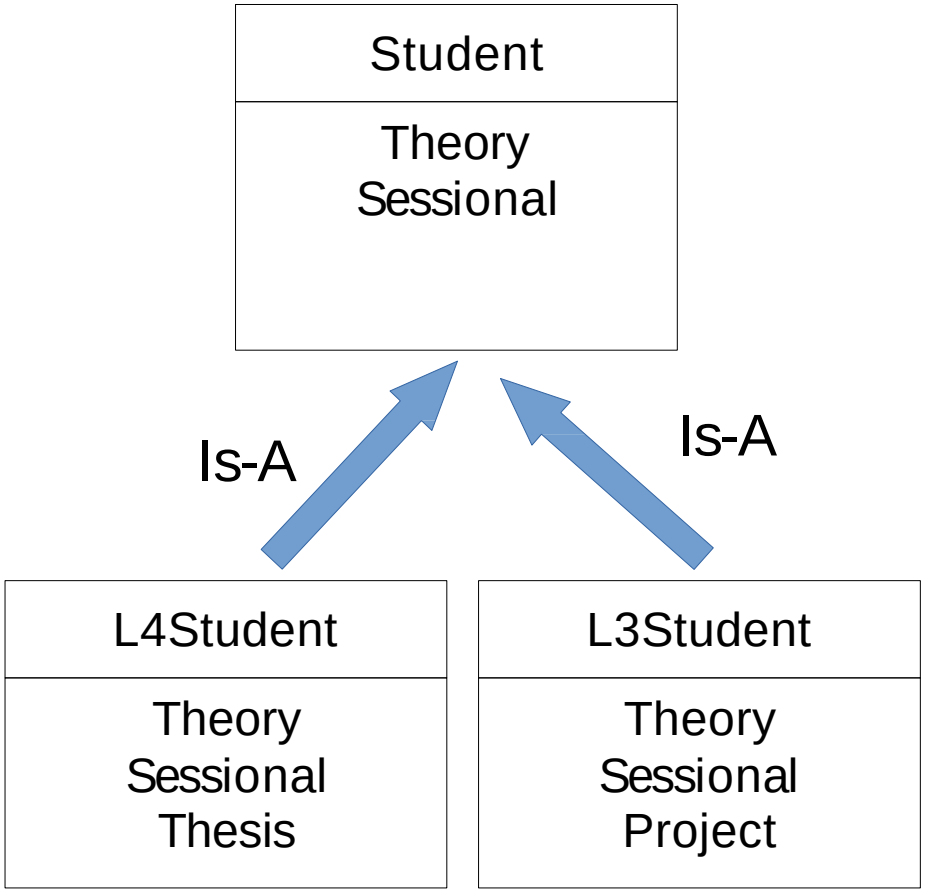
(1/3) Inheritance

Single Inheritance



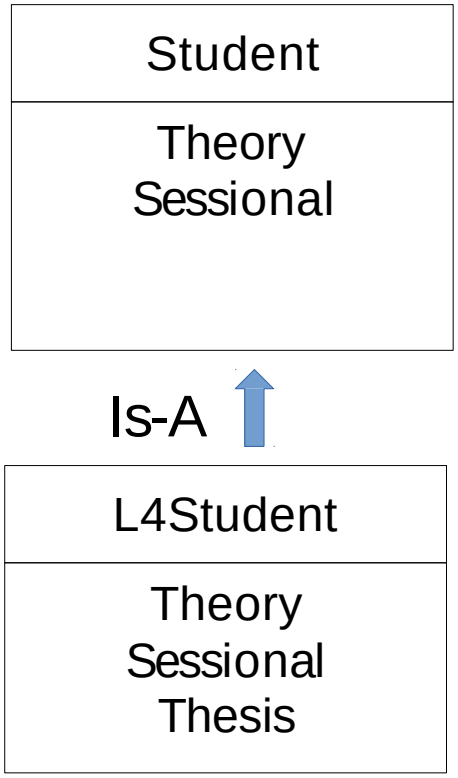
(1/3) Inheritance

Hierarchical Inheritance



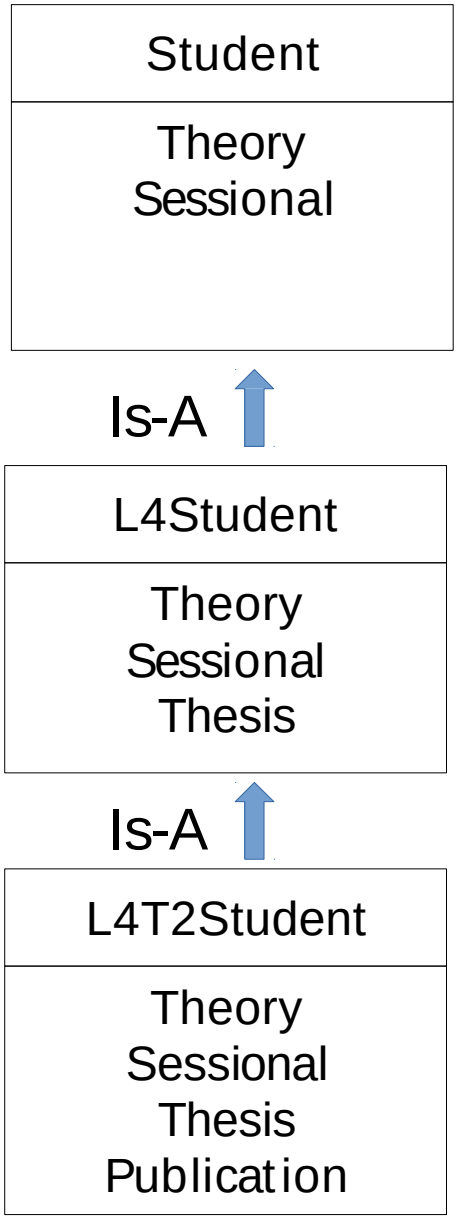
(1/3) Inheritance

Multi-level Inheritance



(1/3) Inheritance

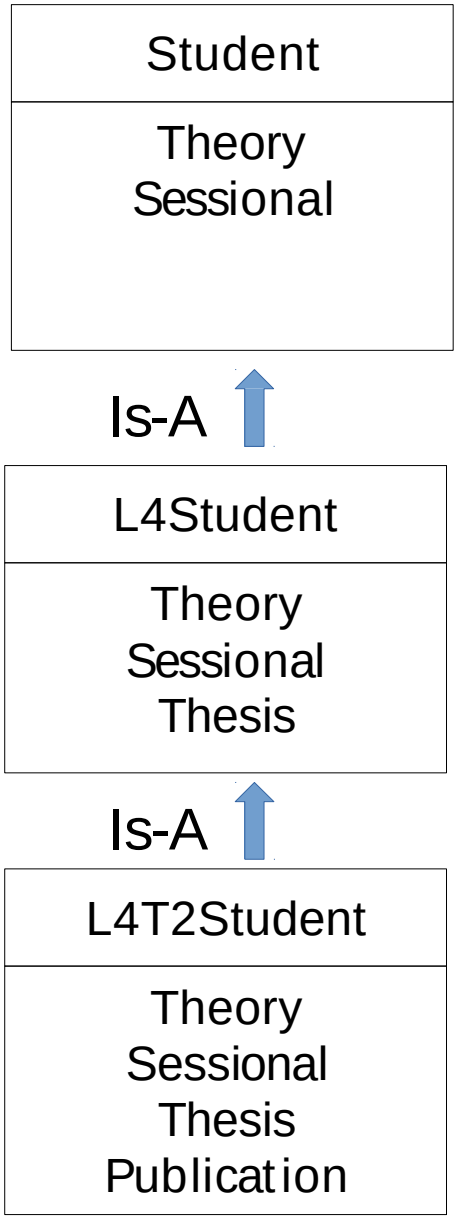
Multi-level Inheritance



(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;
    }
}
```

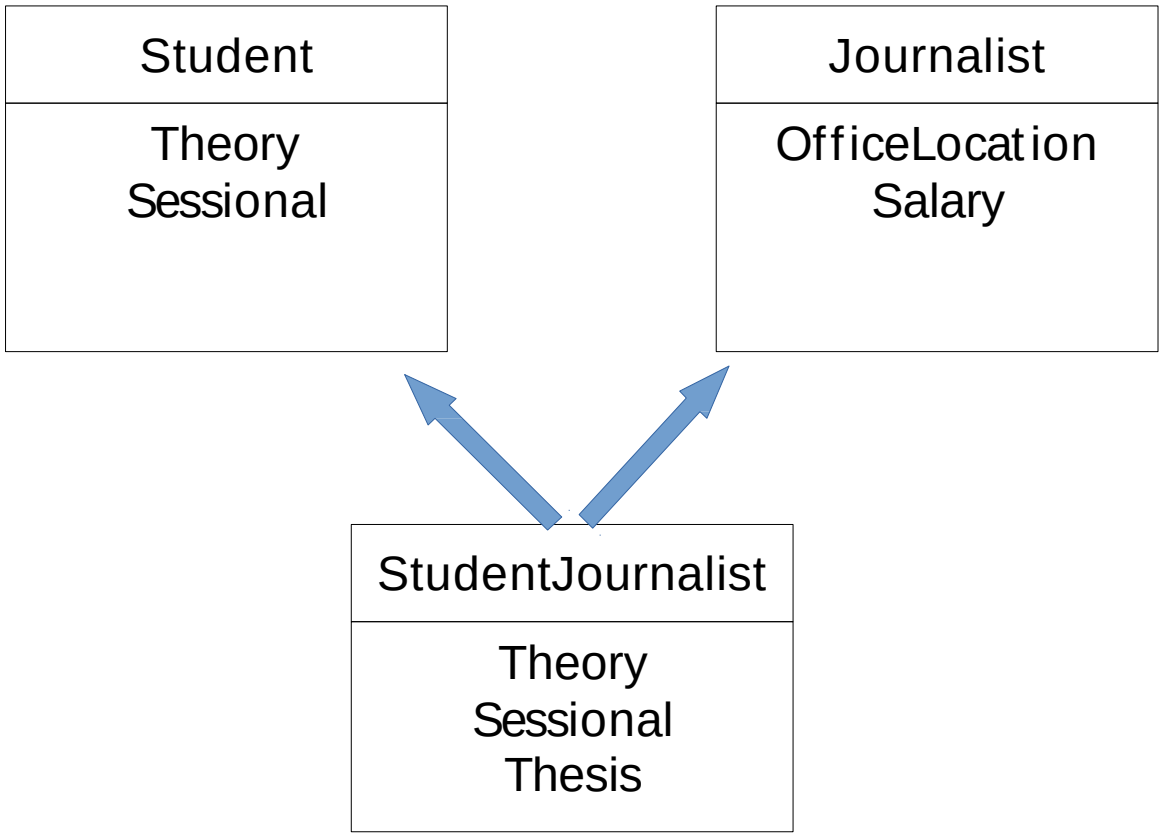


(1/3) Inheritance

Multiple Inheritance?

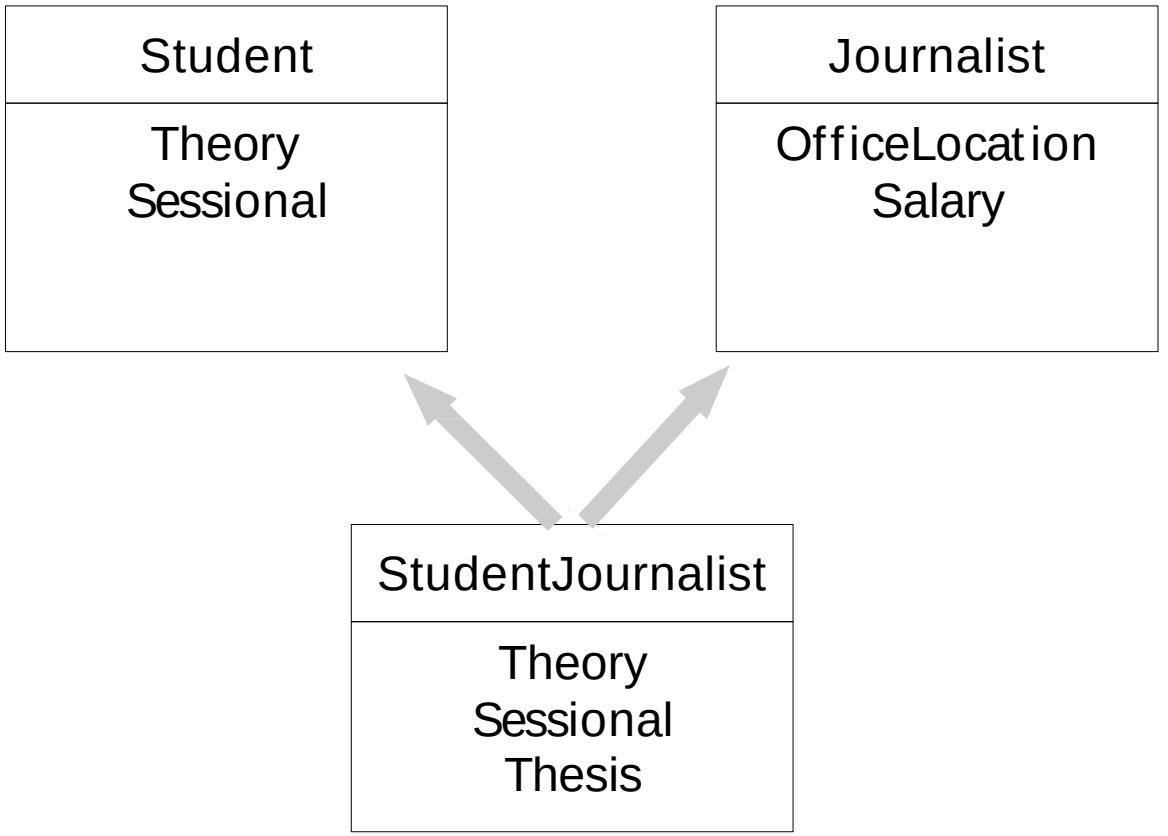
(1/3) Inheritance

Multiple Inheritance?

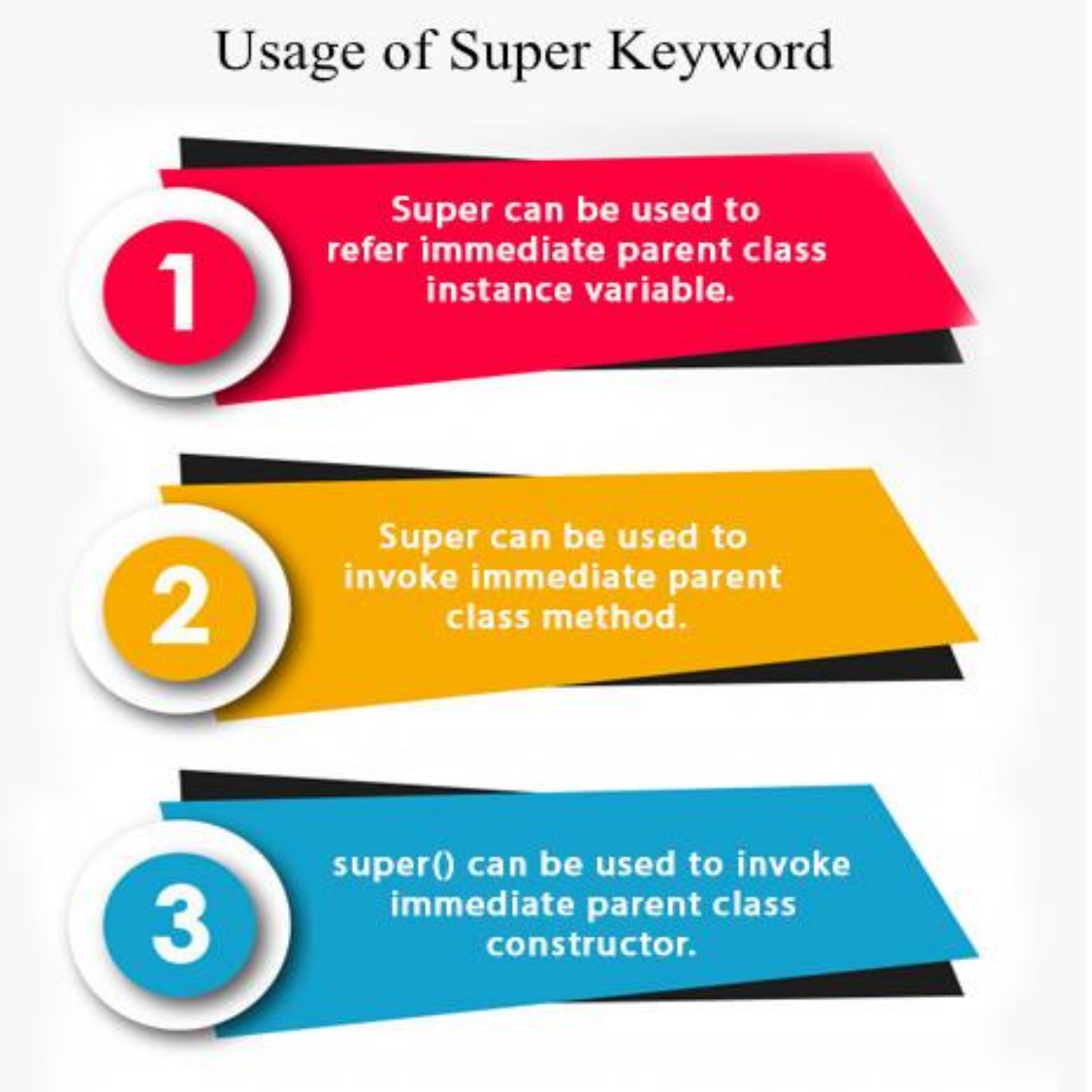


(1/3) Inheritance

Multiple Inheritance? Not allowed in **classes** for java.



Super keyword

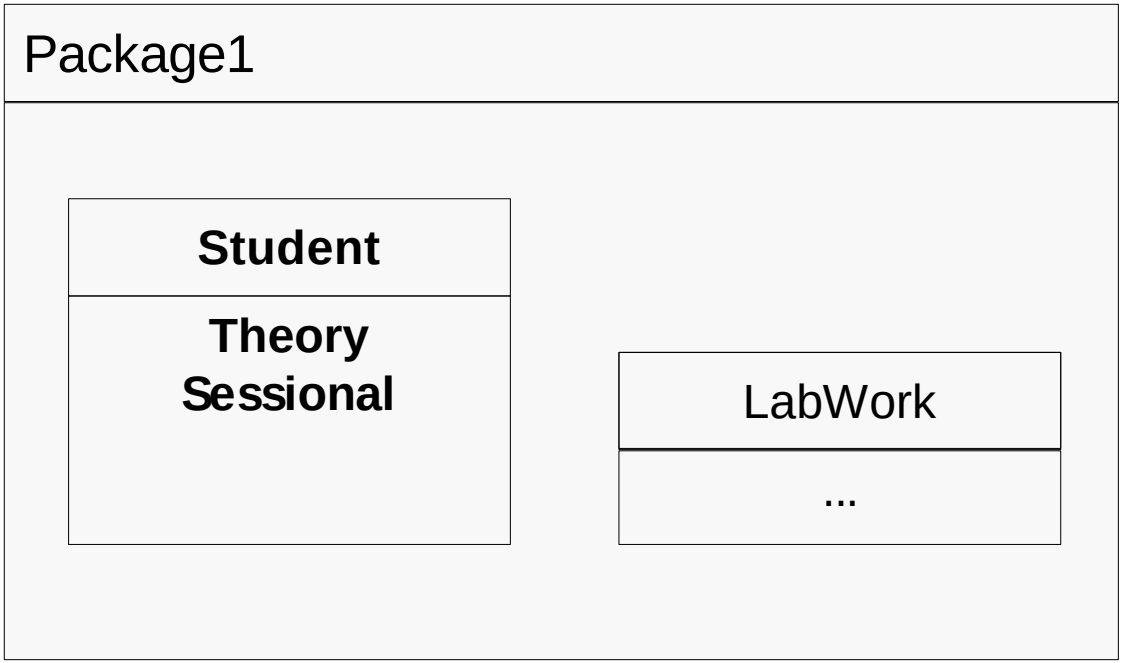


(2/3) Encapsulation

Hiding data and method from outside world

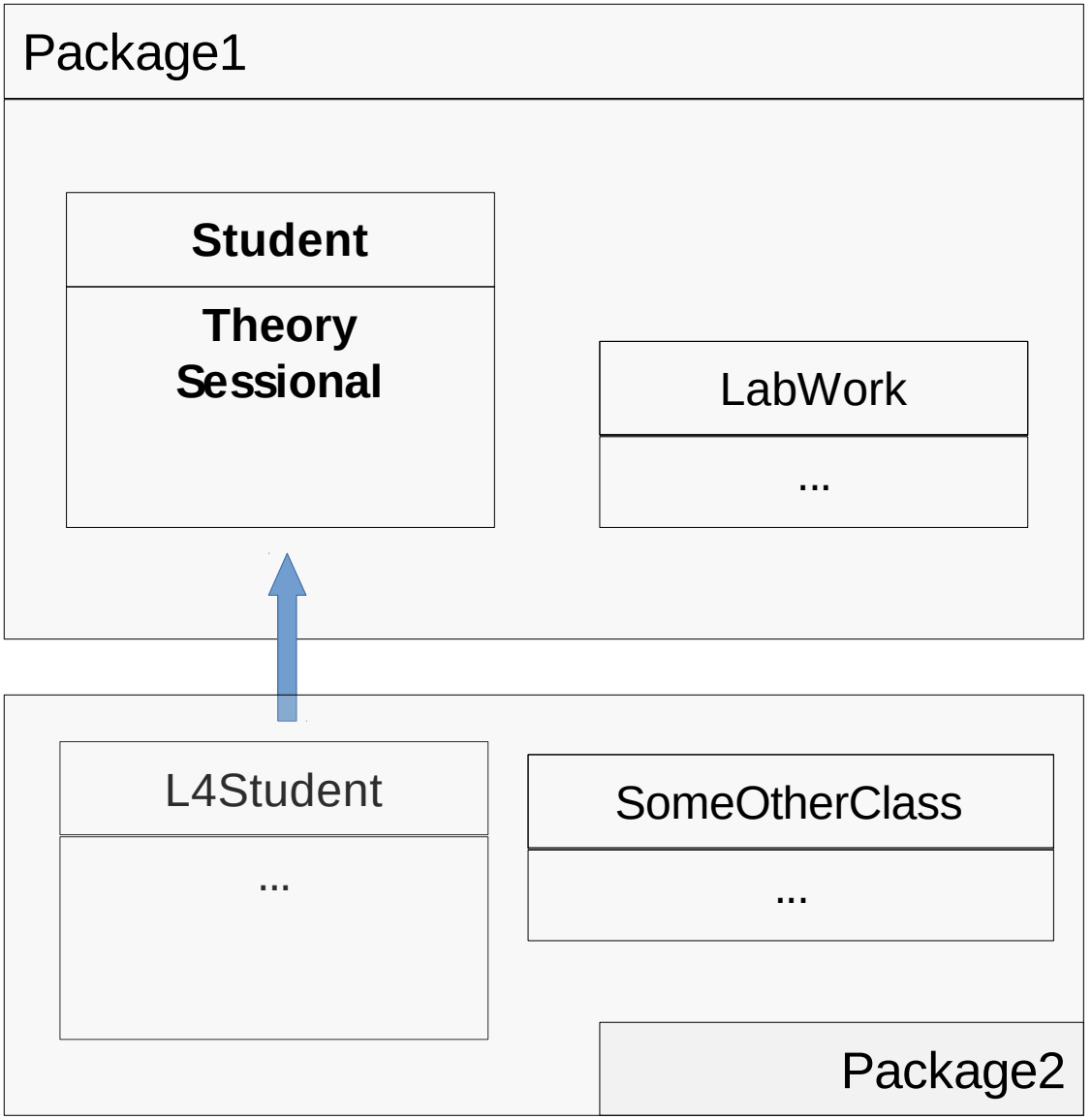
(2/3) Encapsulation

Hiding data and method from outside world



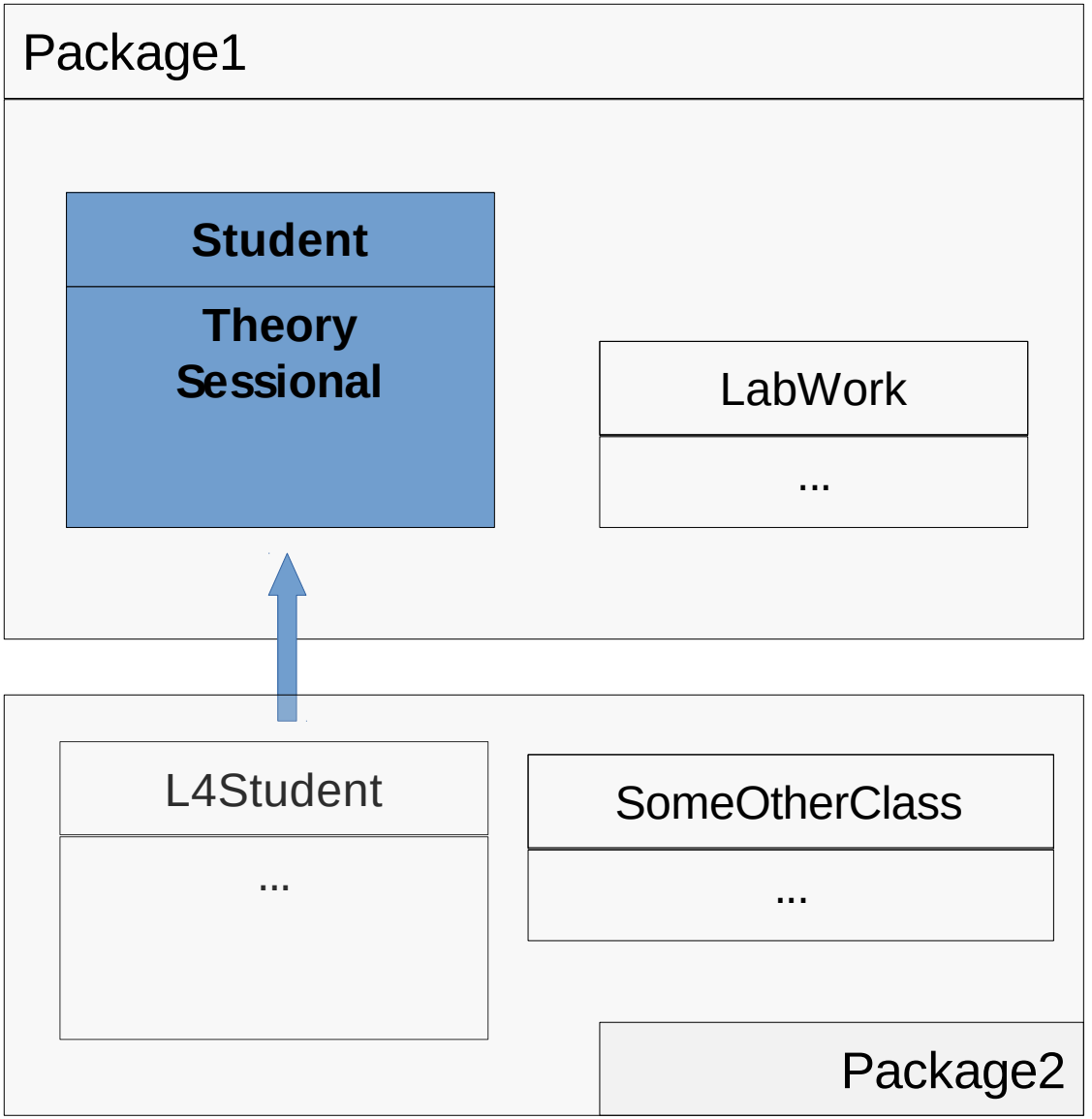
(2/3) Encapsulation

Hiding data and method from outside world



(2/3) Encapsulation

Hiding data and method from outside world

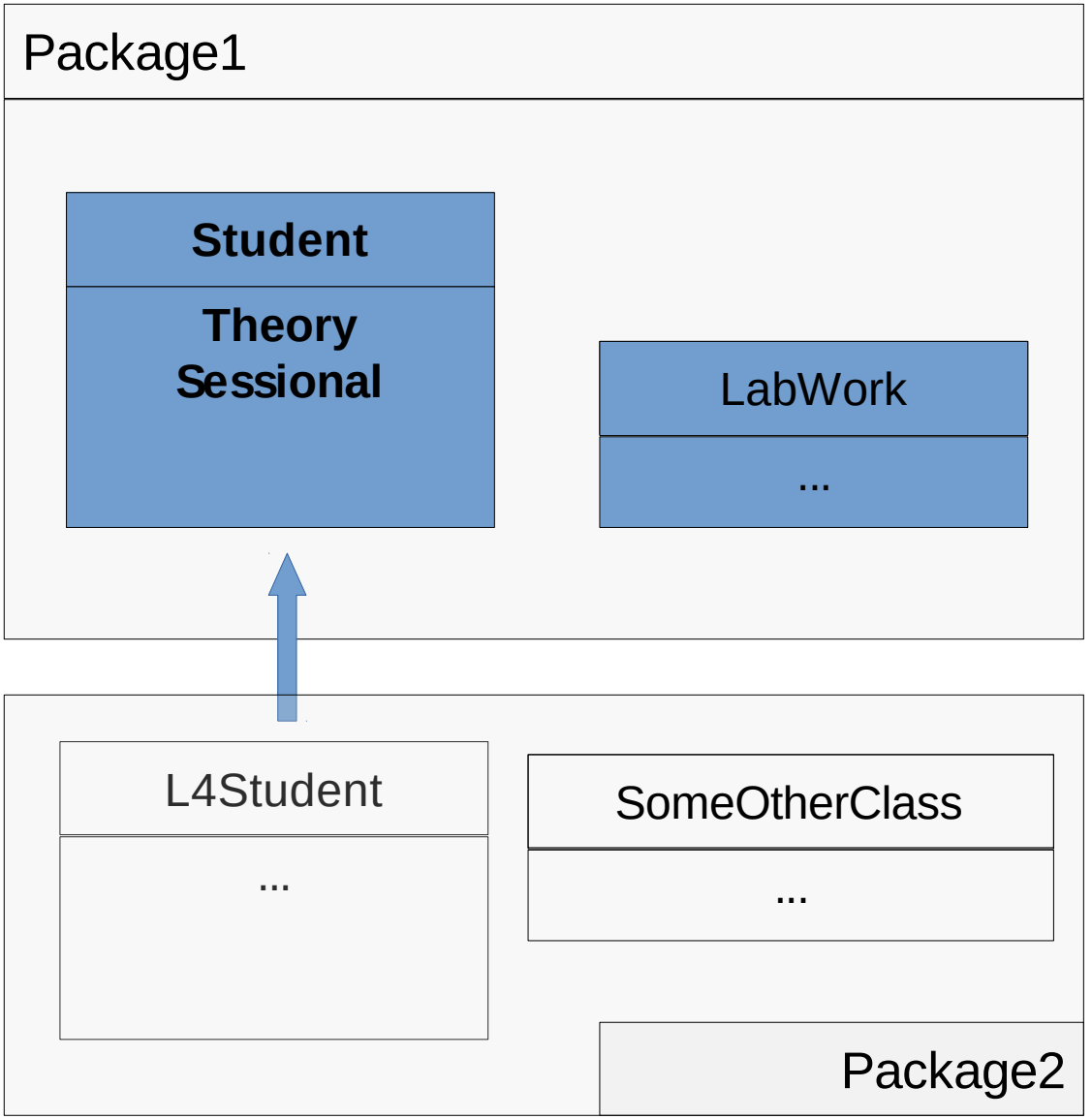


Access Modifier

Private

(2/3) Encapsulation

Hiding data and method from outside world

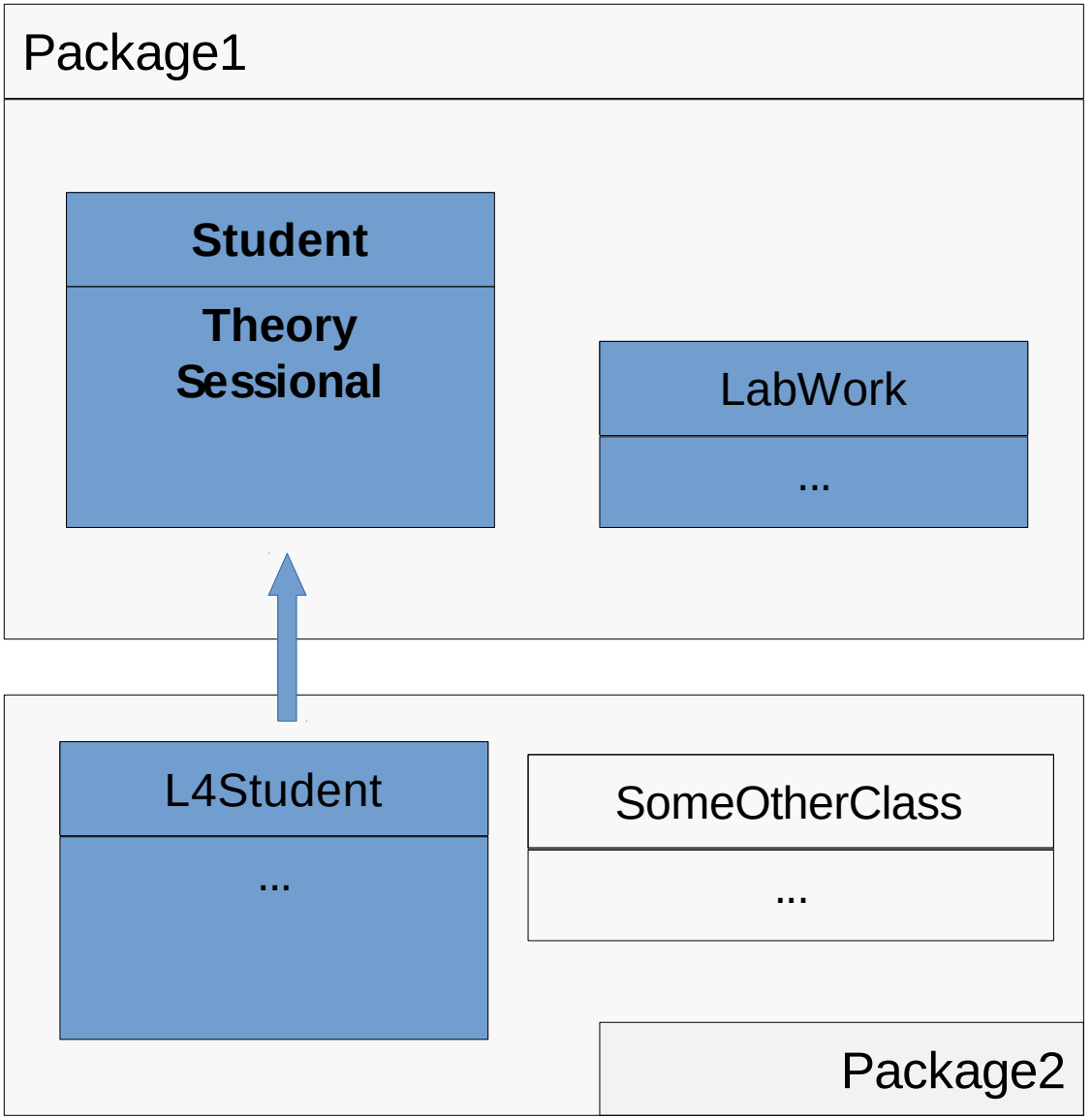


Access Modifier

No Modifier (Package)

(2/3) Encapsulation

Hiding data and method from outside world

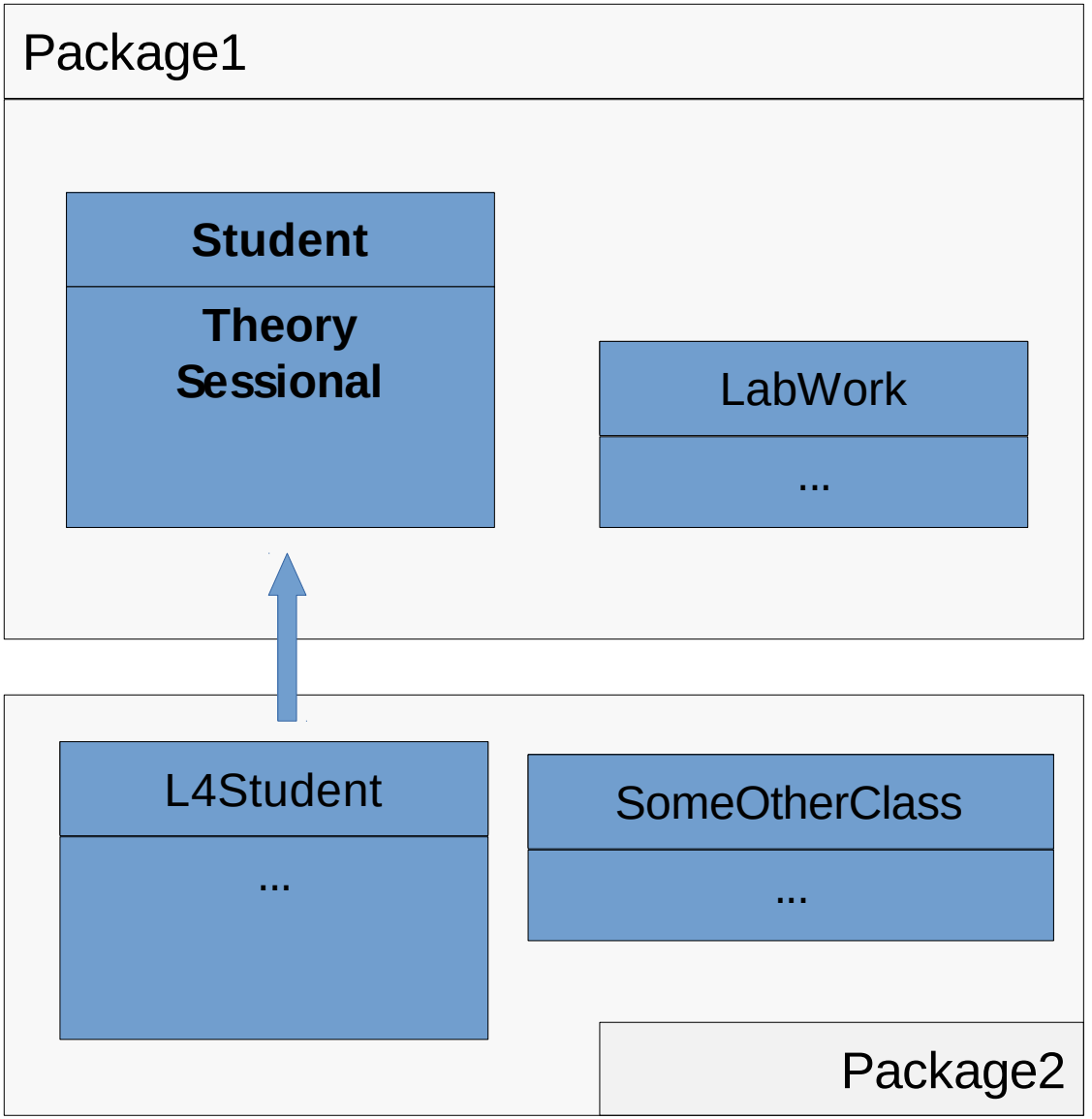


Access Modifier

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

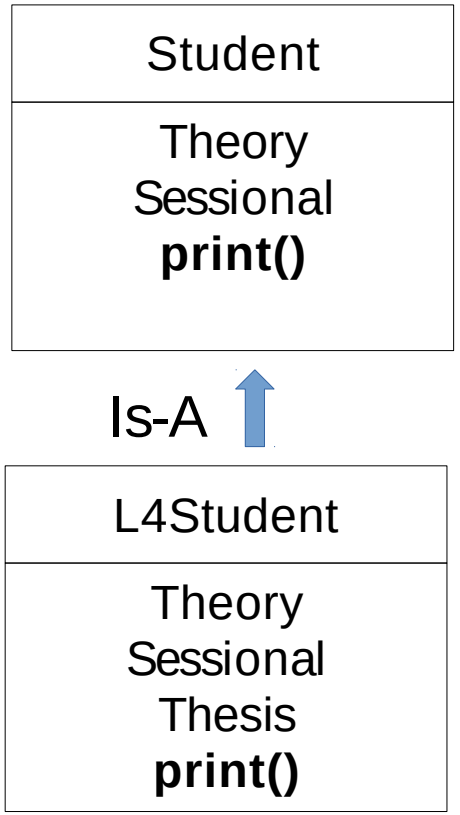
(3/3) Polymorphism

One object/method taking multiple forms

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

(3/3) Polymorphism

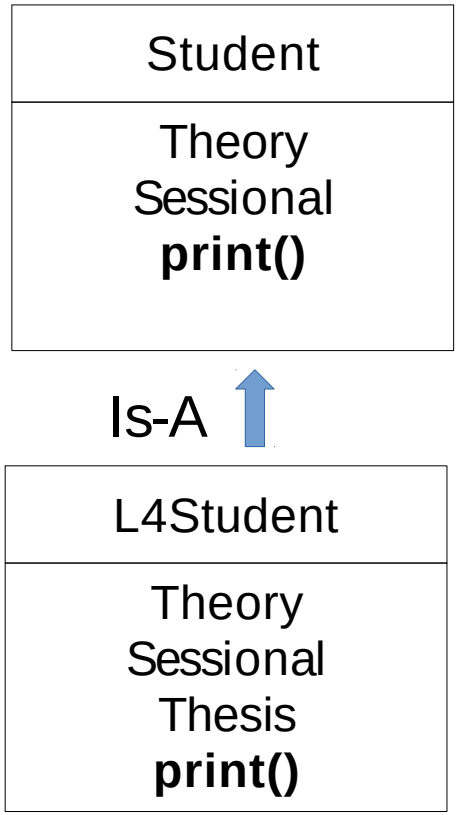
One object/method taking multiple forms



(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();
}
```

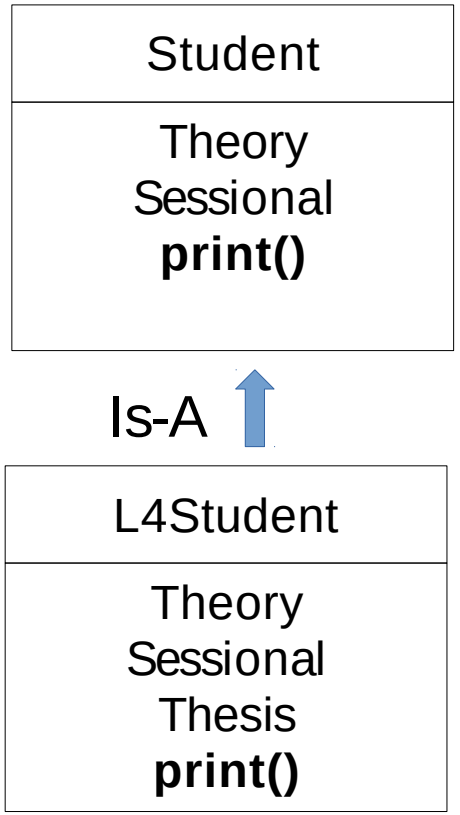


(3/3) Polymorphism

One object/method taking multiple forms

Dynamic Method Distpatch

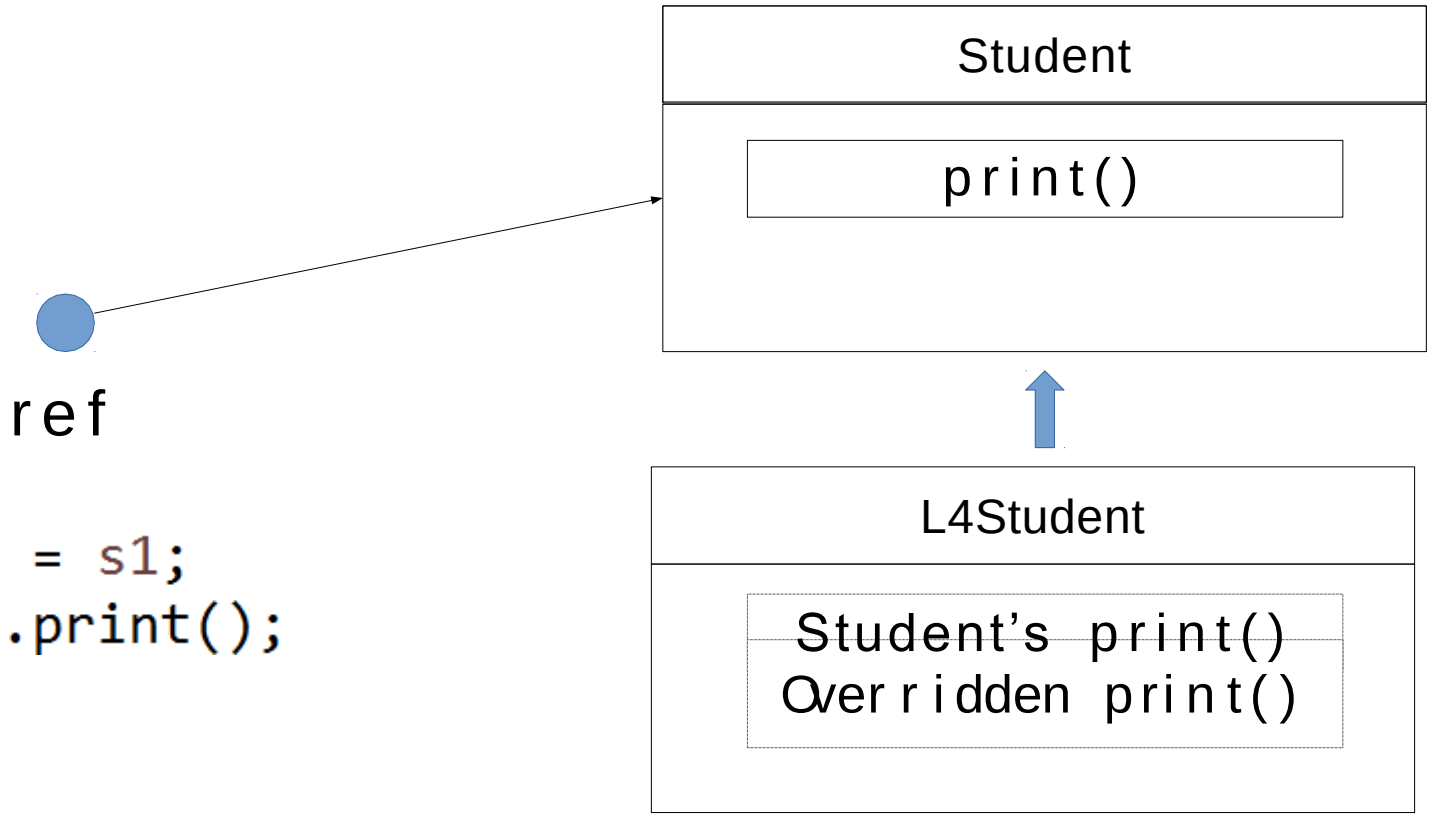
```
public static void main(String[] args) {
    Student s1 = new Student(10, 20);
    L4Student s2 = new L4Student(10, 20, 30);

    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?

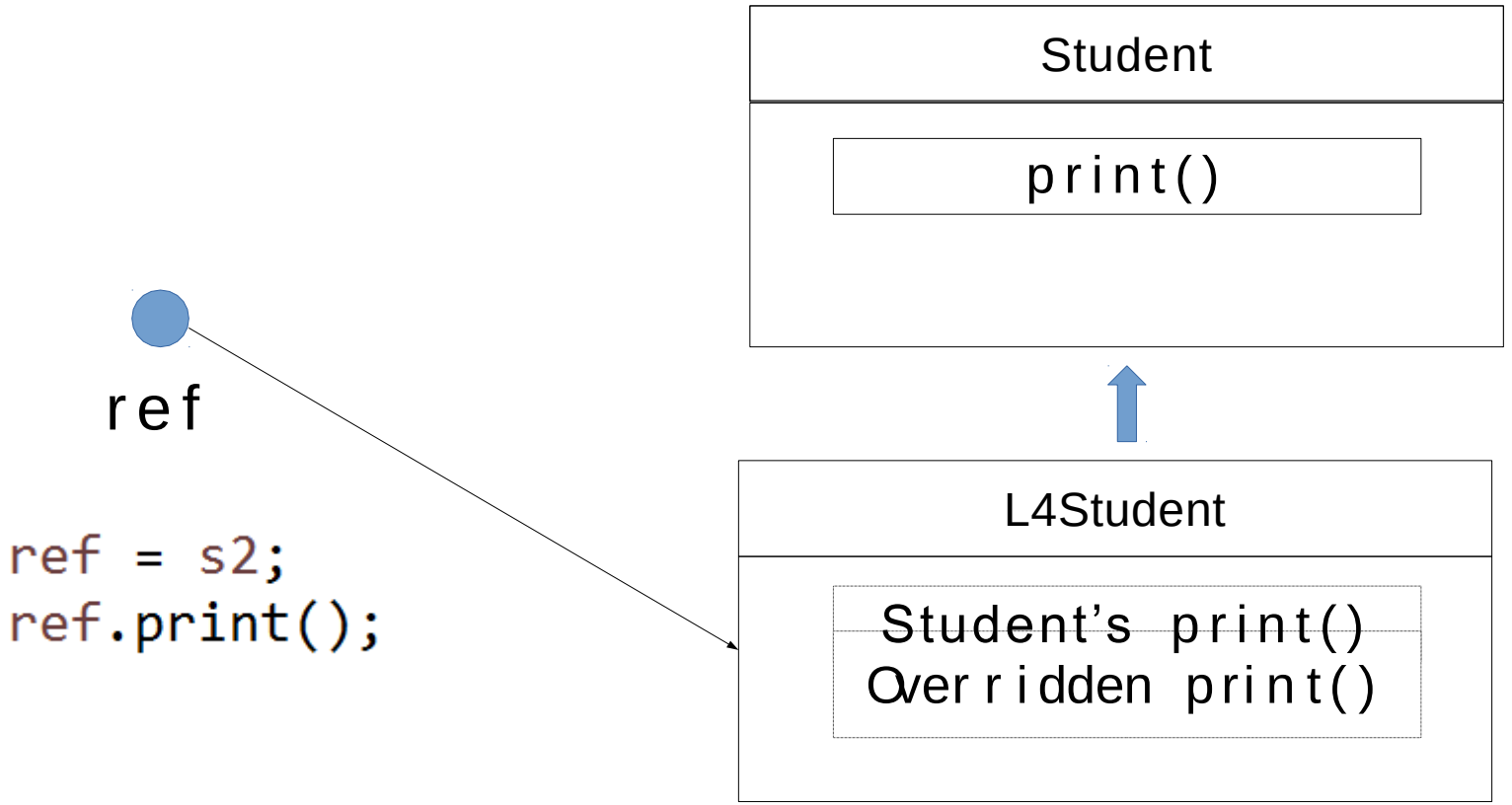



```
ref = s1;
ref.print();
```

(3/3) Polymorphism

One object/method taking multiple forms

Dynamic Method Distpatch – What's happening?



(3/3) Polymorphism

One object/method taking multiple forms

Dynamic binding is resolved by looking at object, during runtime. That's why it is called run-time polymorphism.

```
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

Rules for method overriding:

1. **Final methods can not be overridden.**
2. **Static methods can not be overridden.**
3. **Private methods can not be overridden.**
4. **Constructors cannot be overridden.**
5. **Invoking overridden method from sub-class :** We can call parent class method in overriding method using super keyword.
6. **You must look for the access modifiers while overriding:** For example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.

(3/3) Polymorphism

One object/method taking multiple forms

Static Binding: Compile time polymorphism

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

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

Static Binding

(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();
}
```