Inheritance (Contd.)

CSE-220,

Tasmiah Tamzid Anannya
Lecturer
Dept of CSE, MIST

Abstract Method and Abstract Class

- An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract double getArea();
```

- If a class includes abstract methods, then the class itself **must be** declared abstract, as in:

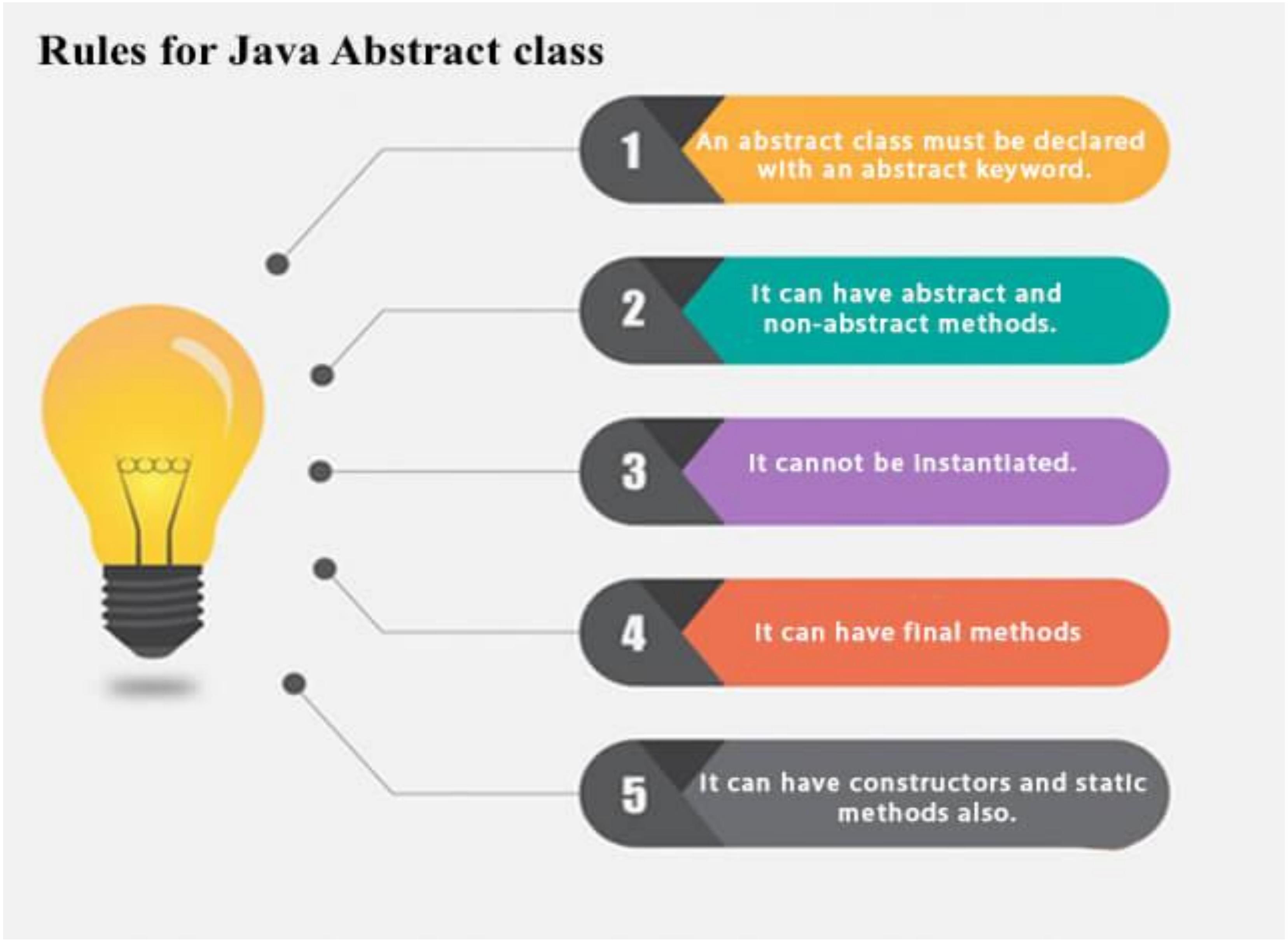
```
public abstract class Shape{
            abstract double getarea();
    }
```

- You cannot create an object of an abstract class.

```
Shape s1=new Shape();   //ERROR
```

- To access the ***abstract*** class, it must be inherited from another class. The subclass usually provides implementations for all of the ***abstract methods*** in its parent class. However, if it does not, then the subclass must also be declared ***abstract***.

Abstract Class



Rules for Java Abstract class

1. An abstract class must be declared with an abstract keyword.
2. It can have abstract and non-abstract methods.
3. It cannot be instantiated.
4. It can have final methods
5. It can have constructors and static methods also.

Abstract Class

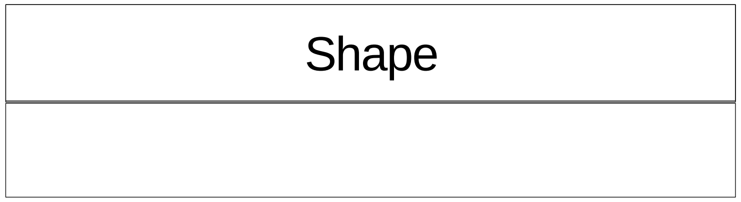
A class that cannot be instantiated, & contains atleast one abstract method

Unimplemented

Abstract Class

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

Unimplemented



Abstract Class

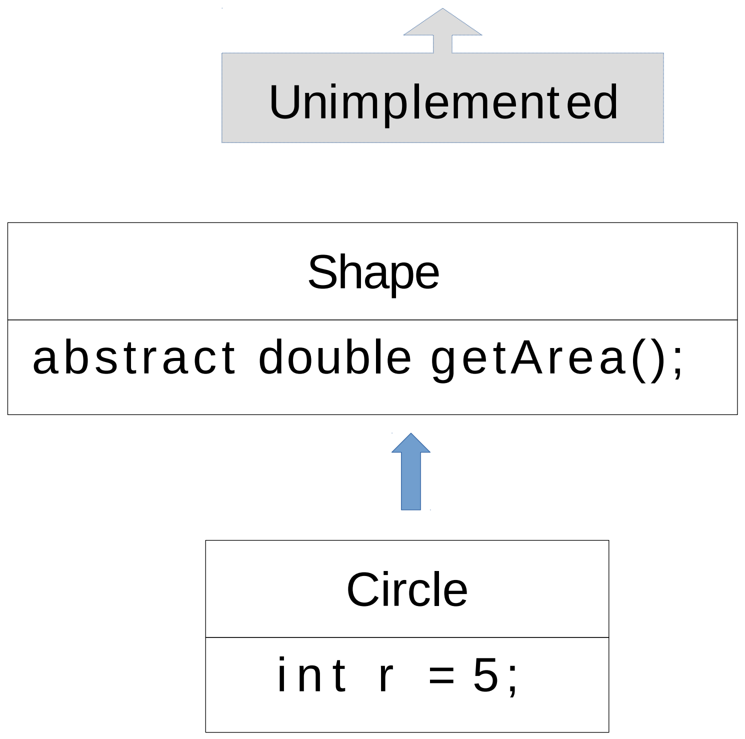
A class that cannot be instantiated, & contains atleast one abstract method

Unimplemented

Shape
abstract double getArea();

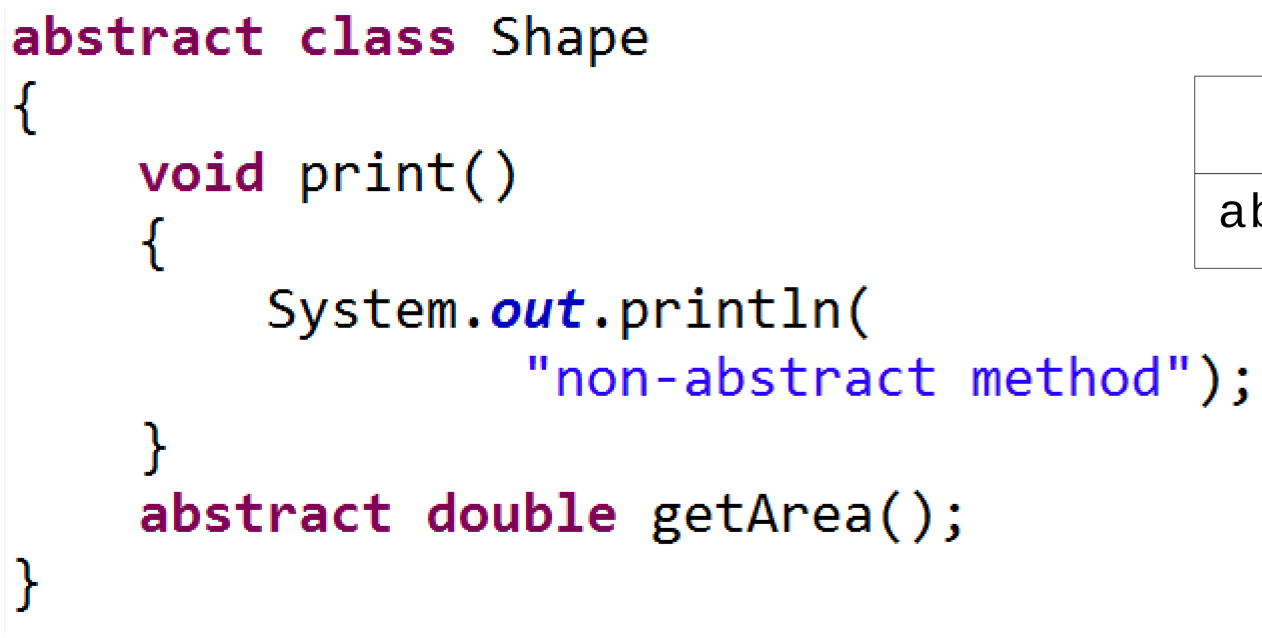
Abstract Class

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

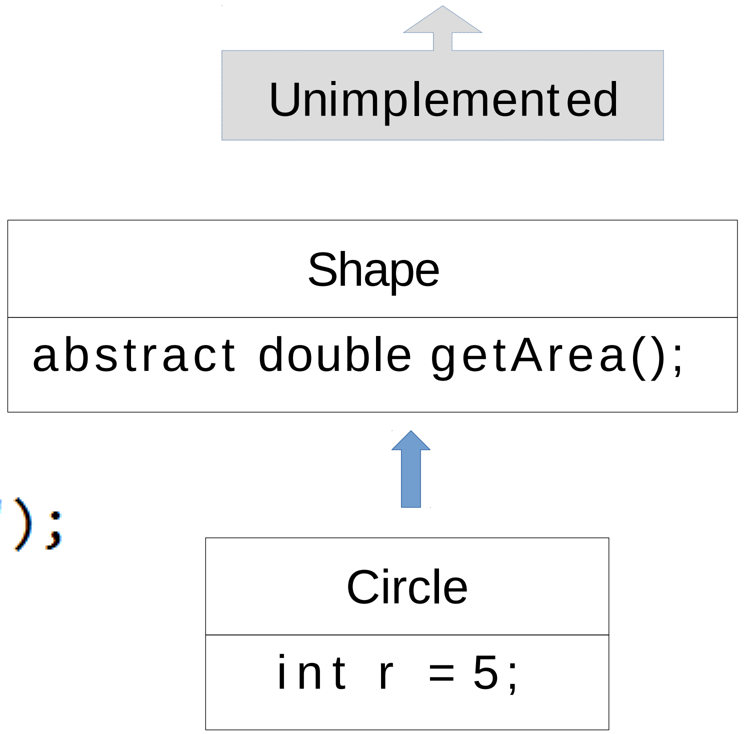


Abstract Class

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();
}
```

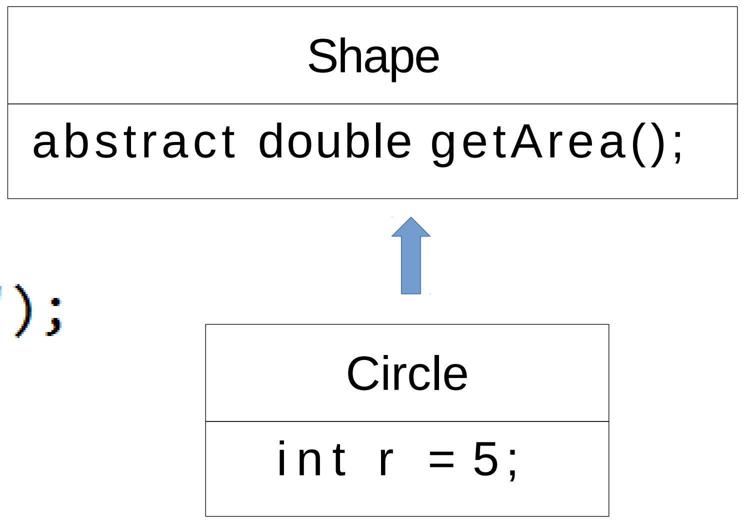


Abstract Class

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

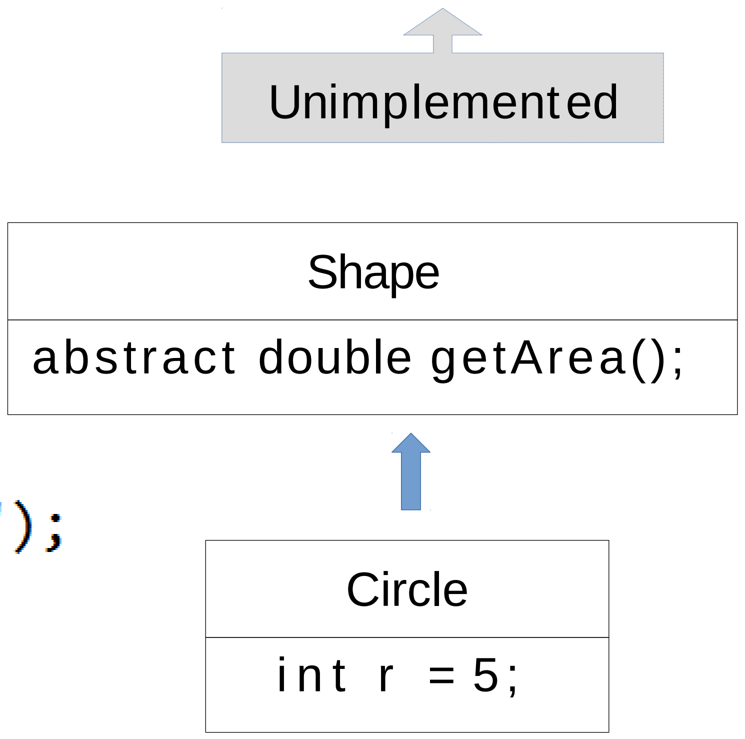
Unimplemented

```
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

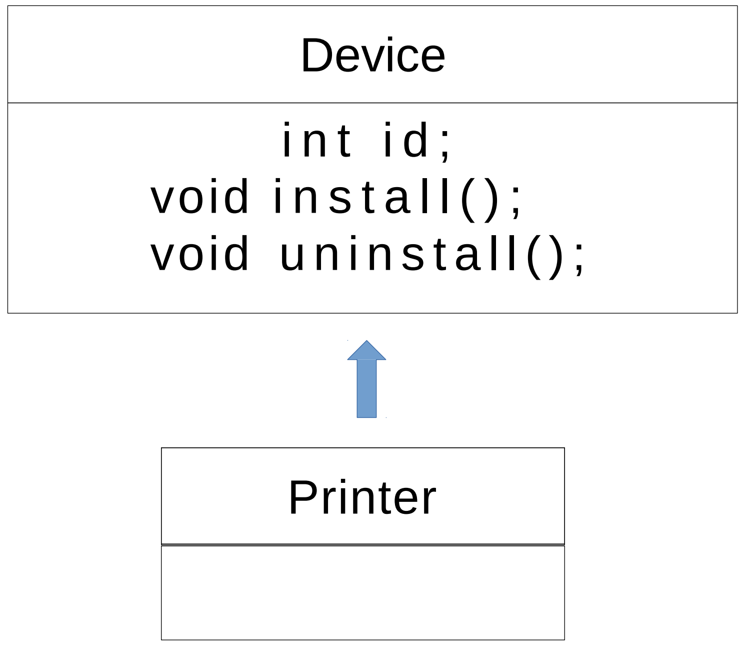


```java
abstract class Shape
{
    void print()
    {
        System.out.println(
                "non-abstract method");
    }
    abstract double getArea();
}
class Circle extends Shape
{
    int r = 5;
    double getArea()
    {
        return 3.1416*r*r;
    }
}
```

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

Interface

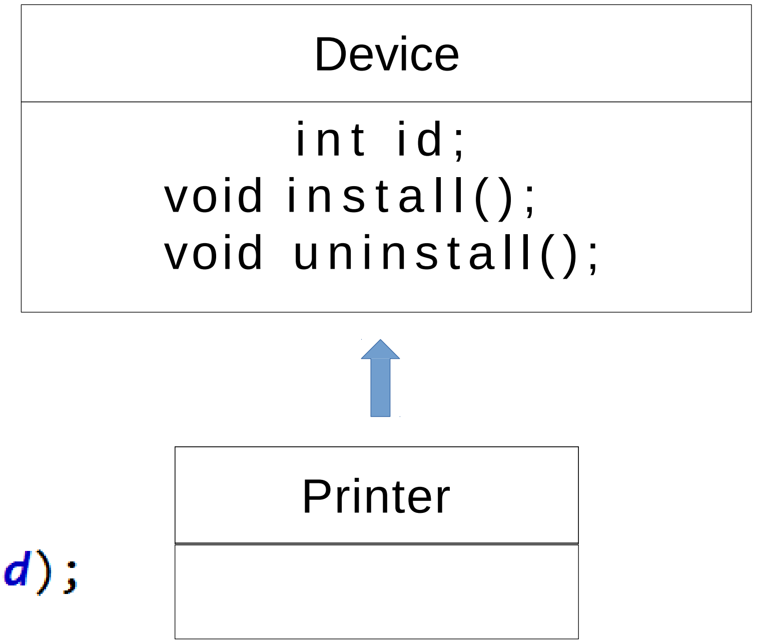
A class that cannot be instantiated, & all it’s methods are abstract



Interface

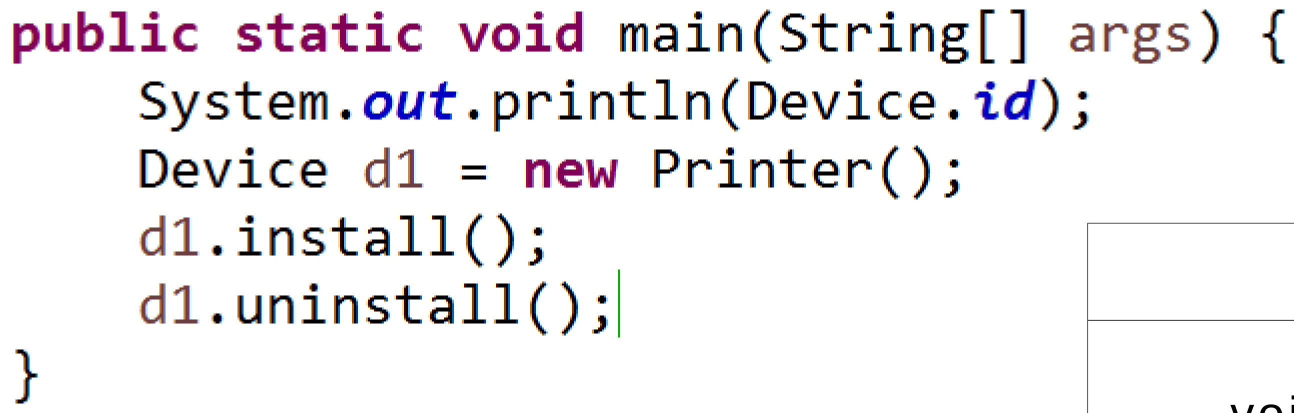
A class that cannot be instantiated, & all it’s methods are abstract

```
interface 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);
    }
}
```

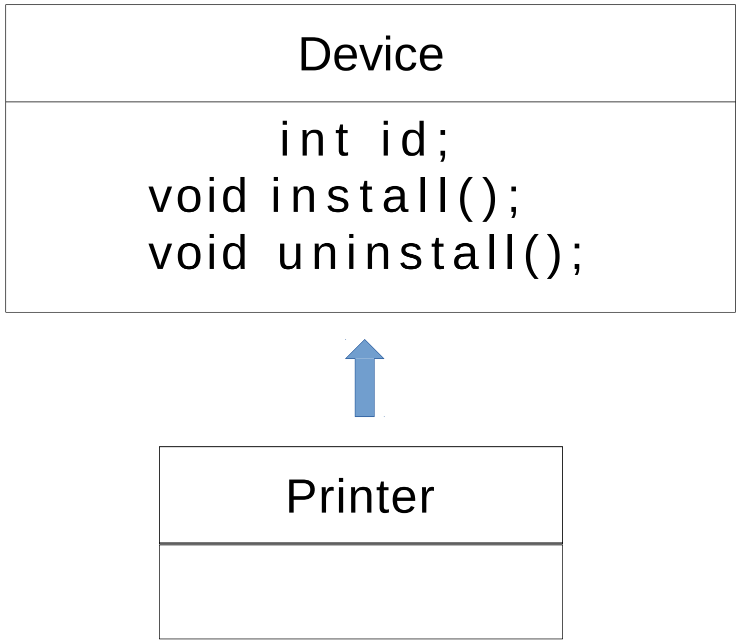


Interface

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();
}
```



Interfaces

An interface is similar to a class in the following ways −

- An interface can contain any number of methods.
- An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a .class file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

Interface

- A class that cannot be instantiated, & all it’s methods are abstract.
- Like **abstract classes**, interfaces **cannot** be used to create objects.
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default abstract and public
- Interface attributes are by default public, static and final
- An interface cannot contain a constructor (as it cannot be used to create objects)

1/3) Inheritance Typecasting

- Most important concepts which basically deals with the conversion of one data type to another datatype implicitly or explicitly.

- The objects can also be typecasted. There are only two types of objects
  - parent object and
  - child object.

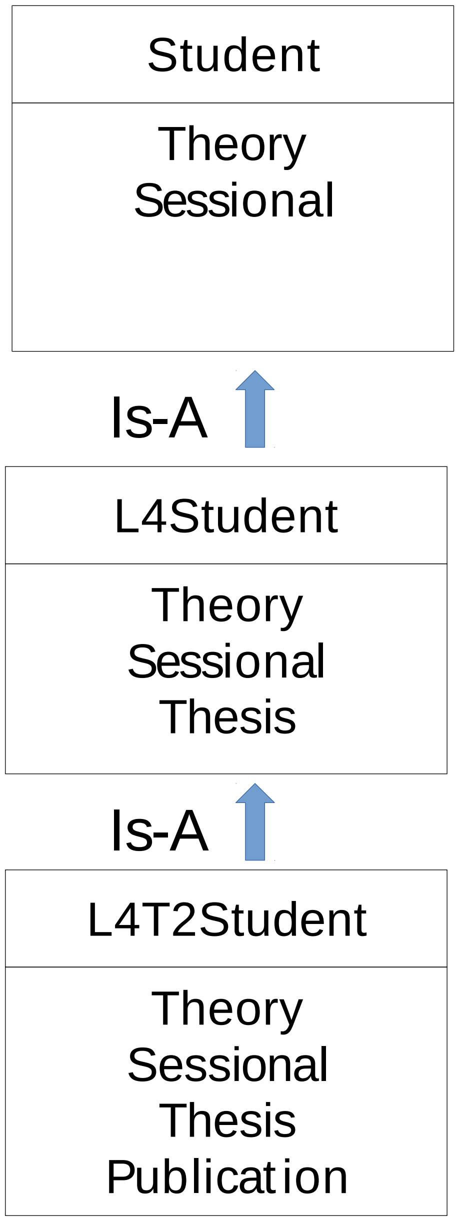
- **Upcasting:**
  - Upcasting is the typecasting **of a child object to a parent object**.
  - can be done implicitly.
  - gives us the flexibility to access the parent class members but it is not possible to access all the child class members using this feature.

- **Downcasting:**
  - Downcasting means the typecasting of a **parent object to a child object**.
  - cannot be done implicitly.

(1/3) Inheritance

Implicit casting (Upcasting)

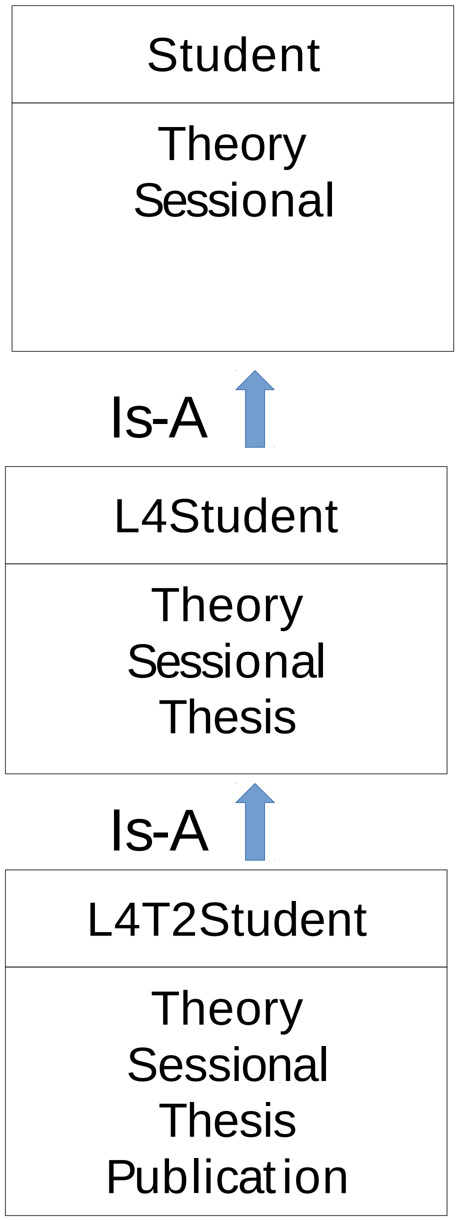
```
public class LabDemo {
    public static void main(String[] args) {
        Student s1 = new L4Student(10,10,10);
    }
}
```



(1/3) Inheritance

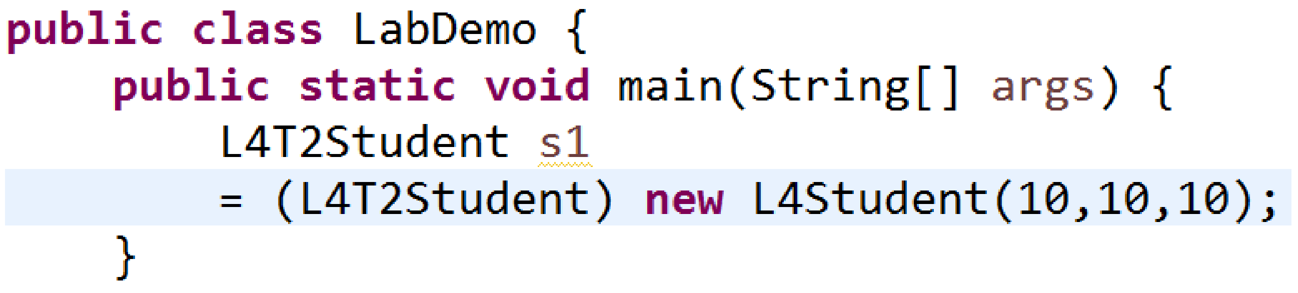
Explicit casting (Downcasting)

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

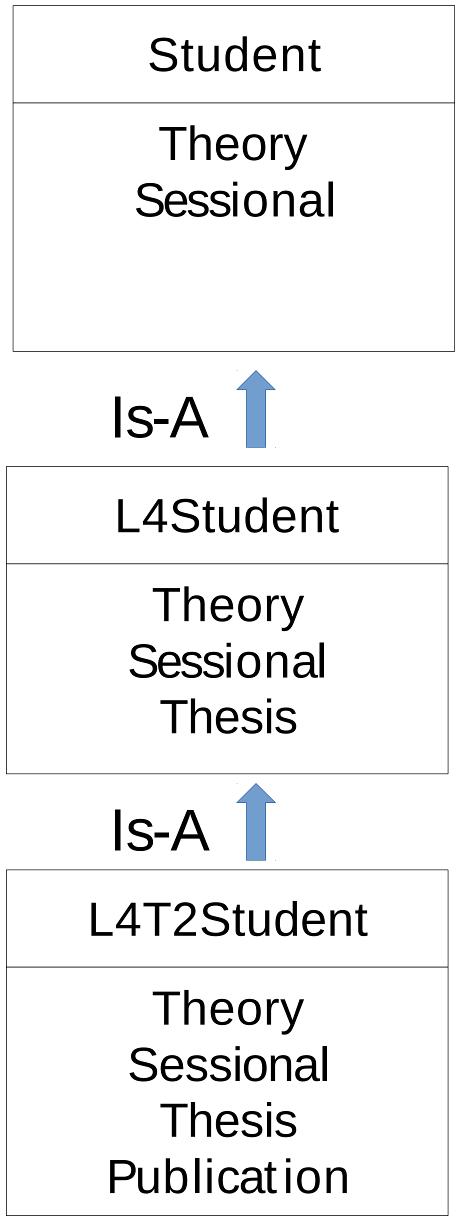


(1/3) Inheritance

Explicit casting (Downcasting)



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



After we define this type of casting explicitly, the compiler checks in the background if this type of casting is possible or not.
If it's not possible, the compiler throws a ClassCastException.

Interface

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

Two things to note:

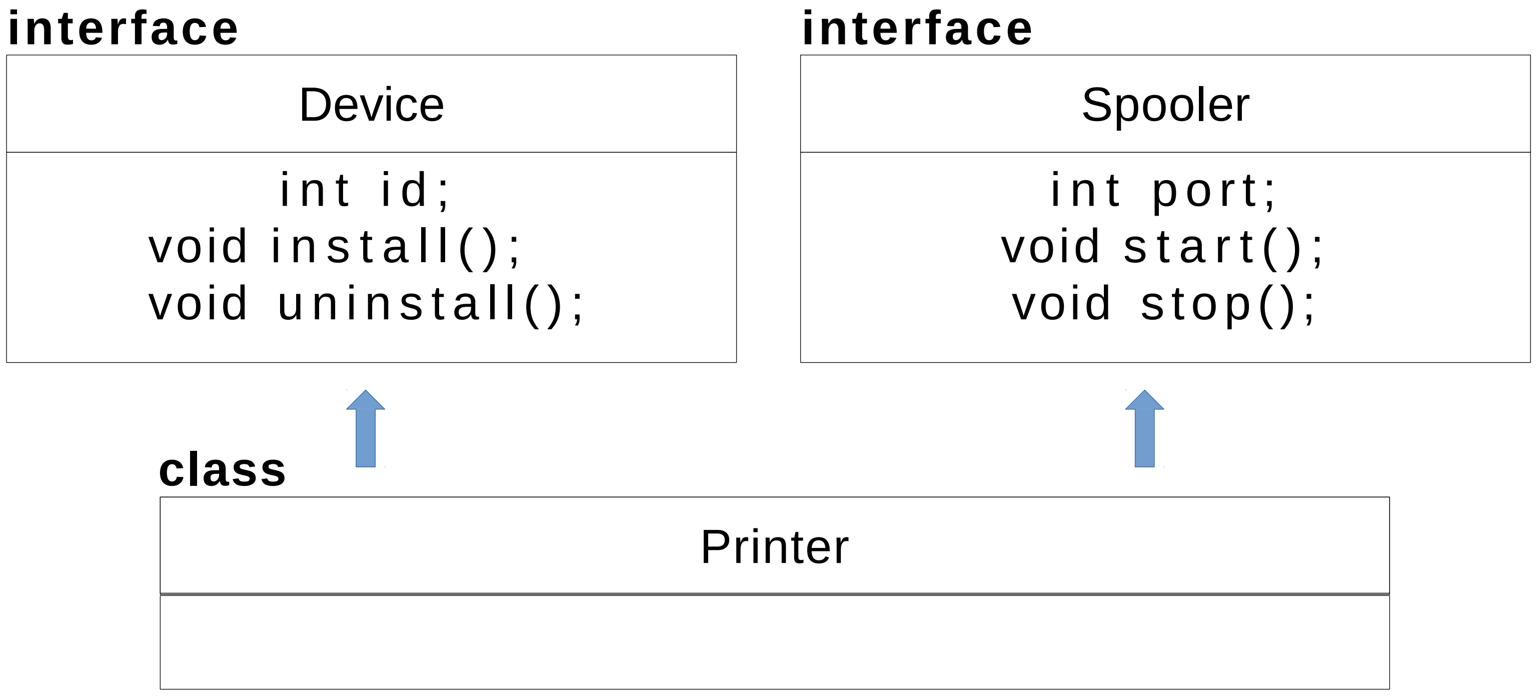
1. Interfaces can have **<u>multiple inheritance</u>.**

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>.**

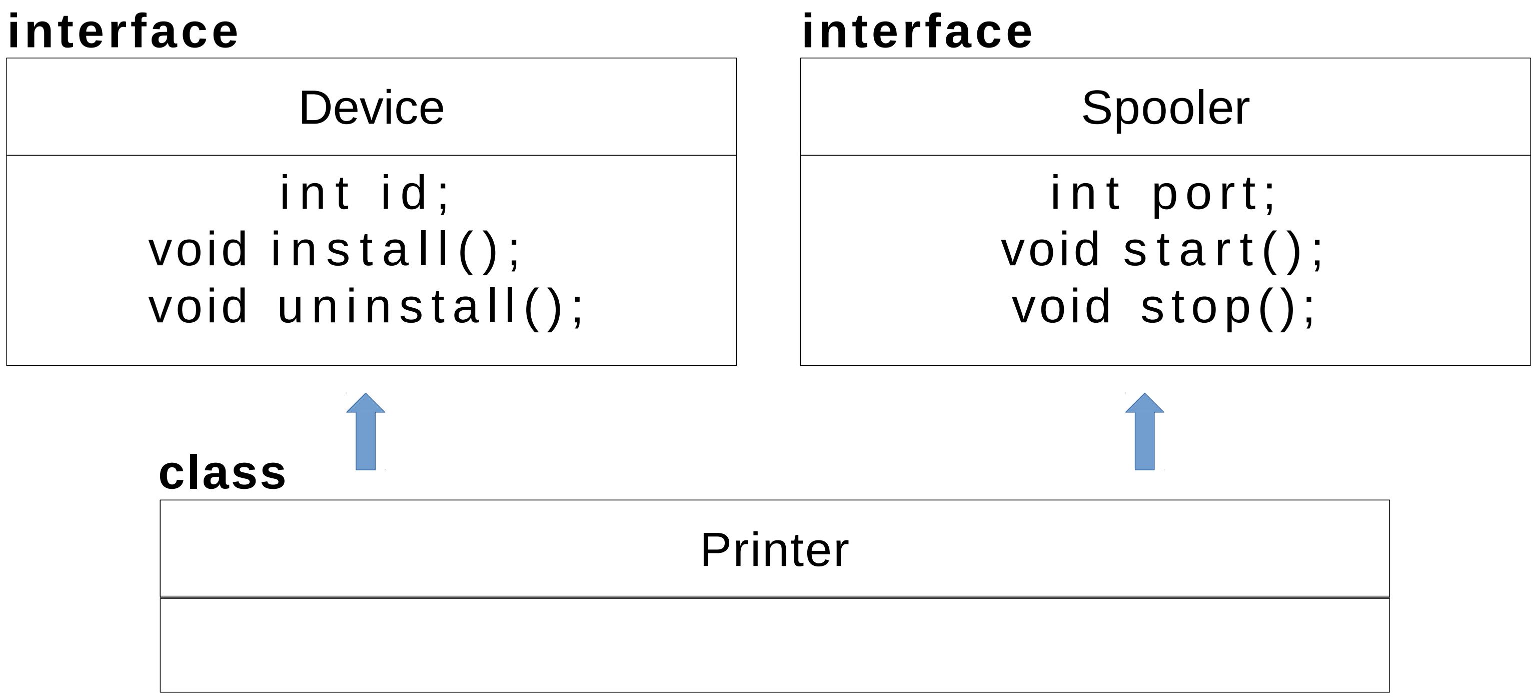


Interface

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

Two things to note:

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



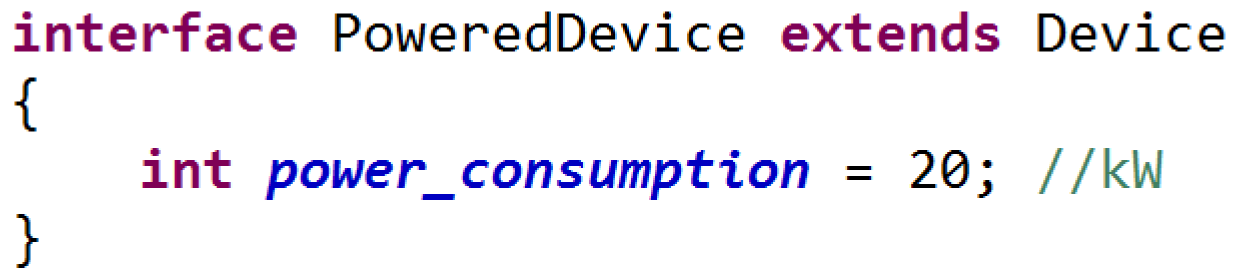
```
class Printer implements Device, Spooler
```

Interface

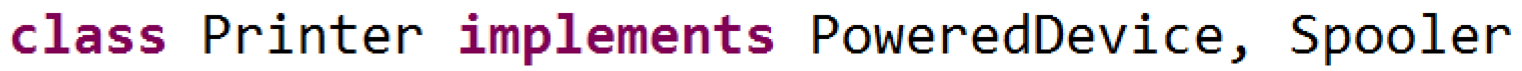
A class that cannot be instantiated, & all it’s methods are abstract

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
```

An interface can extend more than one parent interface.

Interface childInterface extends ParentInterface1, ParentInterface2

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

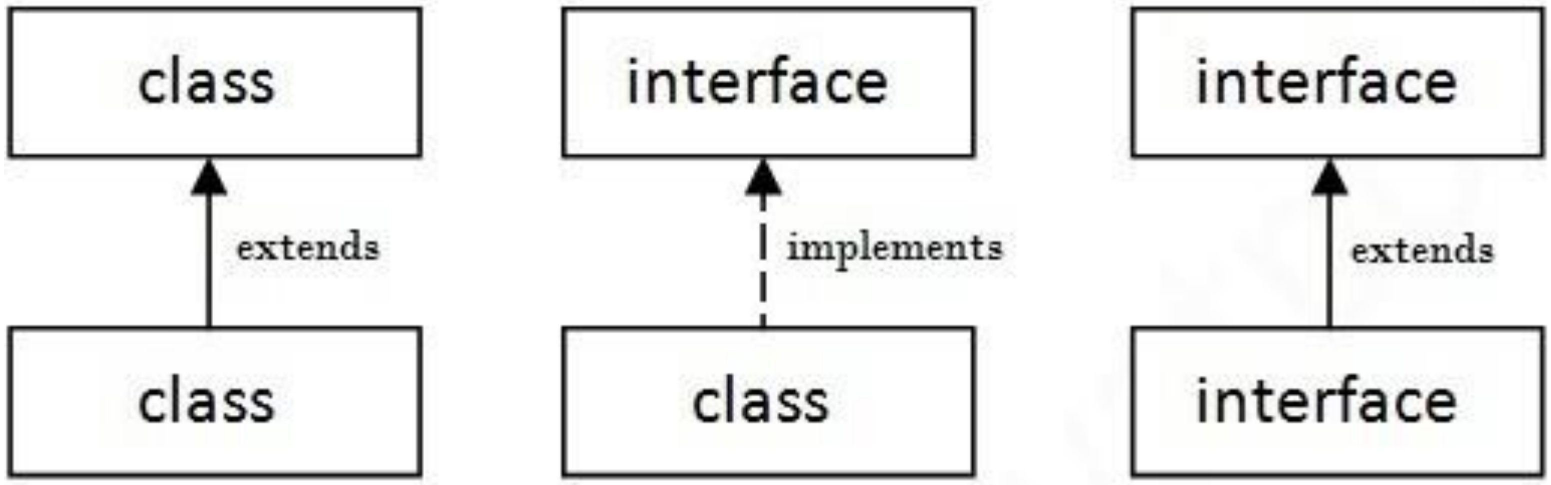


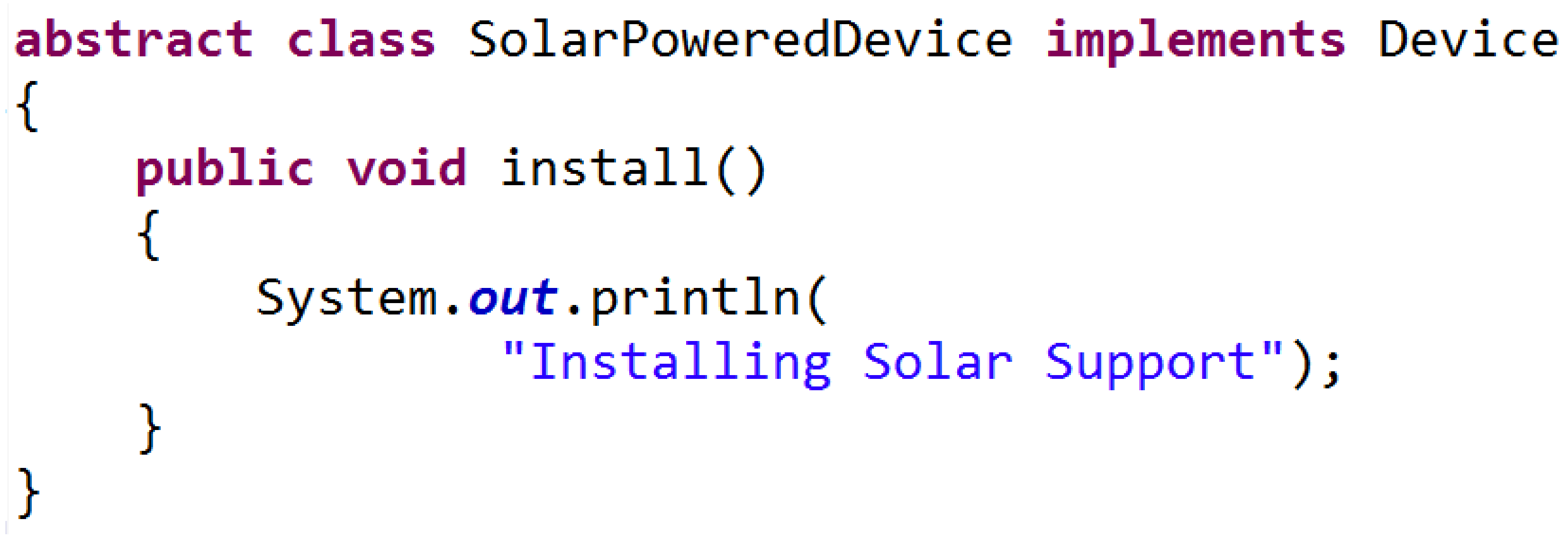
Image Source: https://www.javatpoint.com/interface-in-java

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.



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

1/3) Inheritance Typecasting

- Most important concepts which basically deals with the conversion of one data type to another datatype implicitly or explicitly.

- The objects can also be typecasted. There are only two types of objects
  - parent object and
  - child object.

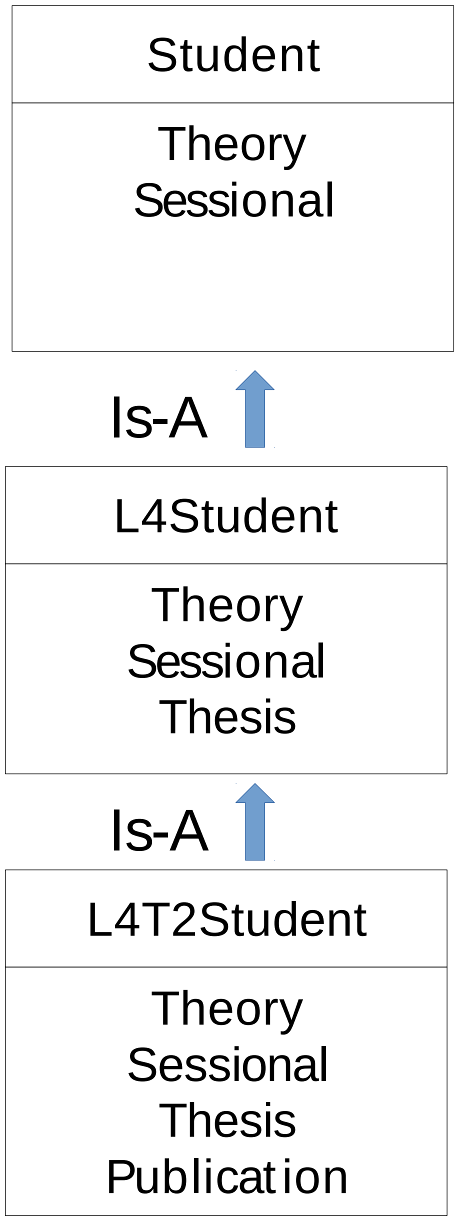
- **Upcasting:**
  - Upcasting is the typecasting **of a child object to a parent object**.
  - can be done implicitly.
  - gives us the flexibility to access the parent class members but it is not possible to access all the child class members using this feature.

- **Downcasting:**
  - Downcasting means the typecasting of a **parent object to a child object**.
  - cannot be done implicitly.

(1/3) Inheritance

Implicit casting (Upcasting)

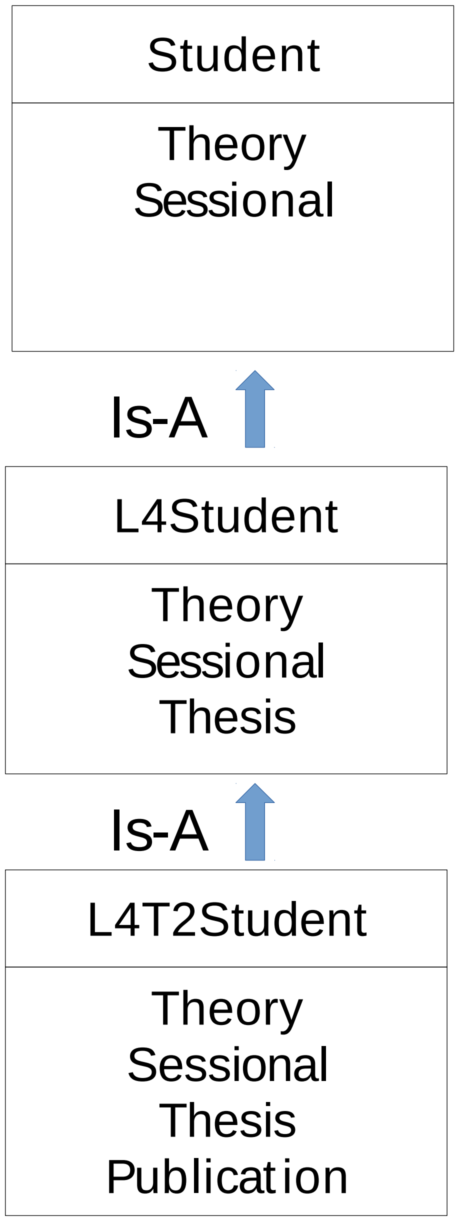
```
public class LabDemo {
    public static void main(String[] args) {
        Student s1 = new L4Student(10,10,10);
    }
}
```



(1/3) Inheritance

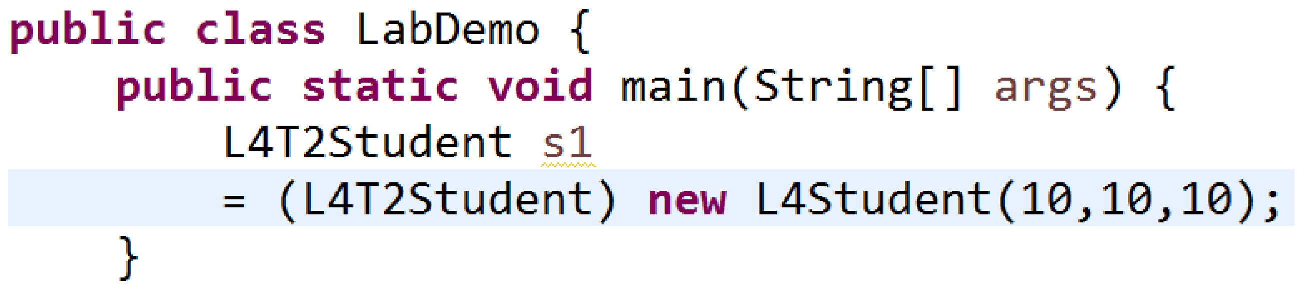
Explicit casting (Downcasting)

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

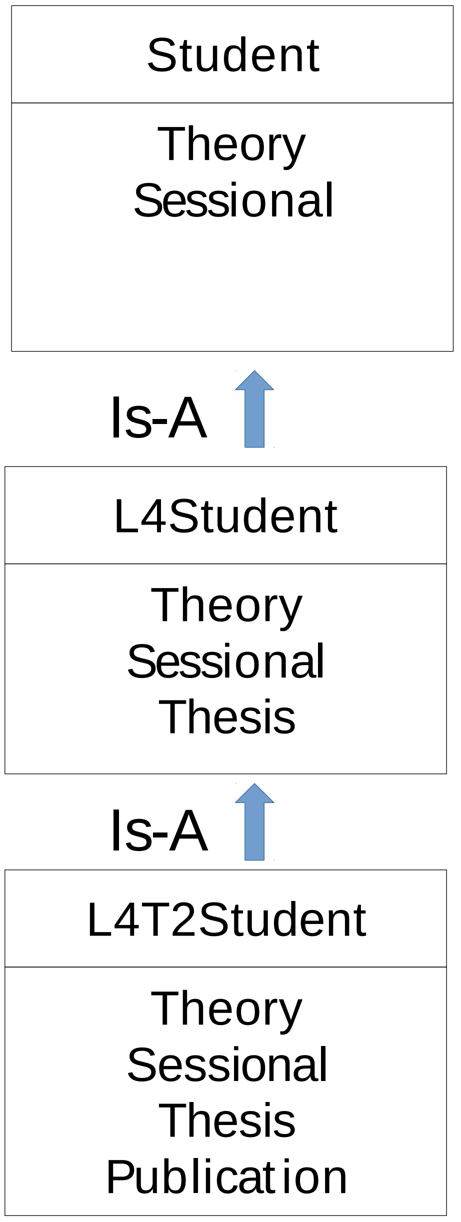


(1/3) Inheritance

Explicit casting (Downcasting)



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



**After we define this type of casting explicitly, the compiler checks in the background if this type of casting is possible or not.
If it's not possible, the compiler throws a ClassCastException.**

So, do the following for downcasting:

```
public static void main(String[] args){
        L4Student s1=new L4T2Student(10, 10, 10,10);
        L4T2Student s2=new L4T2Student(20,20, 20, 20);
        s2= (L4T2Student) s1;
}
```