

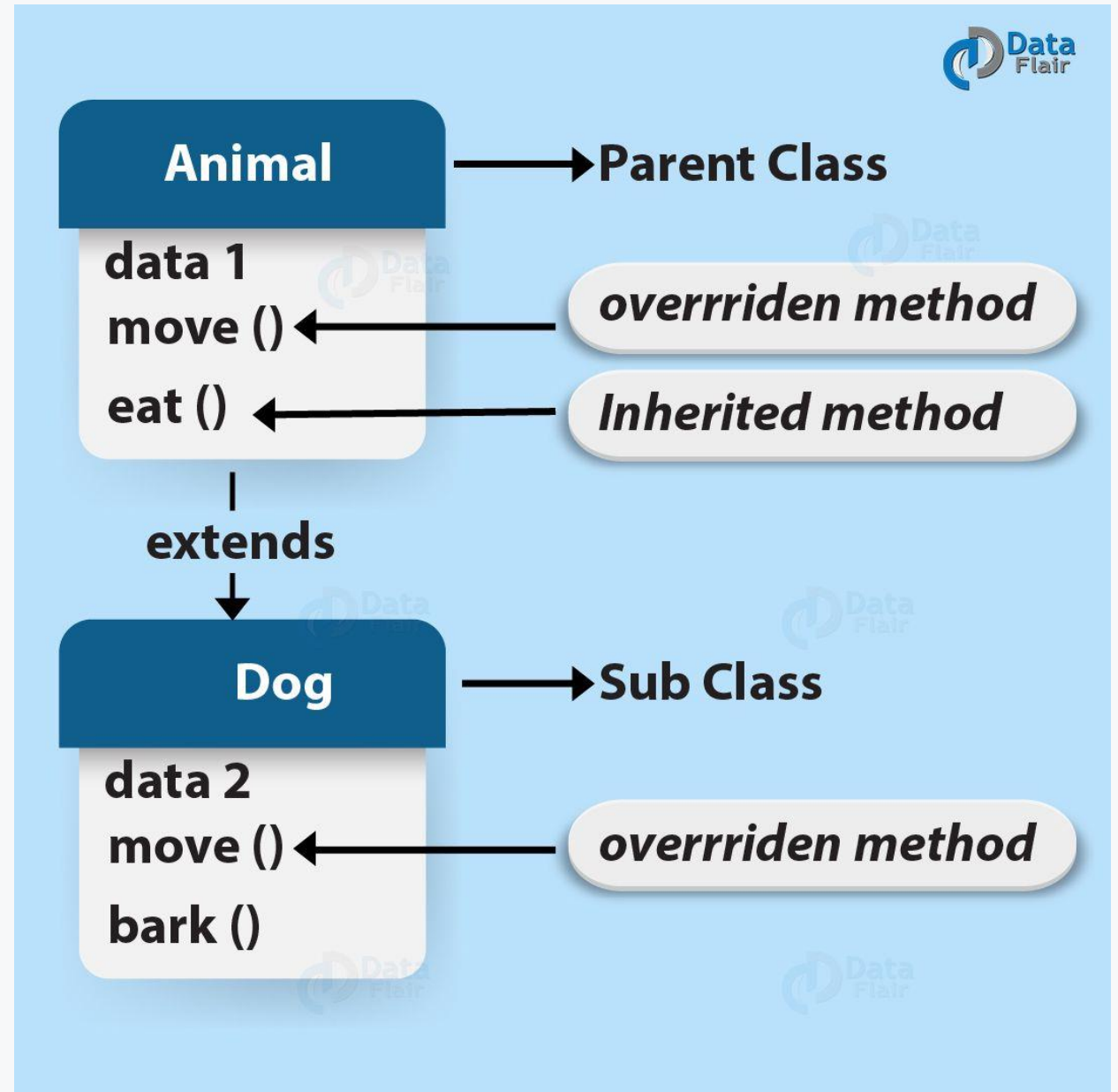
METHOD OVERRIDING



Introduction

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.

When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.



Why Method Overriding?

- We can achieve Run time polymorphism using the concept of method overriding.
- Overridden methods are another way that java implements "one interface , multiple methods" aspect of polymorphism.

Example of Method Overriding

```
class Vehicle{  
    void run(){  
        System.out.println("Vehicle is running");  
    }  
    class Bike extends Vehicle{  
        void run(){  
            System.out.println("Bike is running safely");  
        }  
    }  
    public static void main(String args[]){  
        Bike obj = new Bike();  
        obj.run();  
    }  
}
```

Output is:

Bike is running safely

Note:

- A Superclass Variable Can Reference a Subclass Object.
- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass
- A subclass cannot override method that is declared **final** in super class
- A subclass cannot override method that is declared **static** in super class



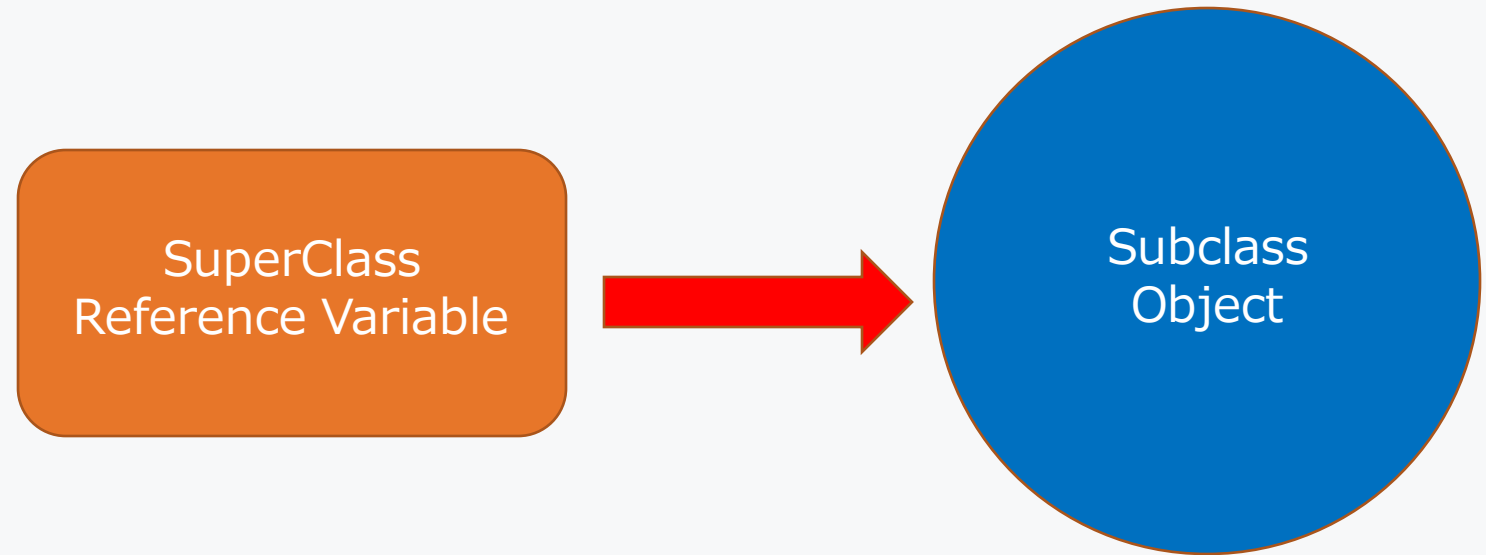
DYNAMIC METHOD DISPATCH



INTRODUCTION

- Method overriding is one of the ways in which Java supports Runtime Polymorphism.
- **Dynamic method dispatch** is the mechanism by which a call to an overridden method is resolved at run time.
- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

A superclass reference variable can refer to a subclass object. This is also known as **upcasting**. Java uses this fact to resolve calls to overridden methods at run time.



Method Overloading v/s Method Overriding

| Method Overloading | Method Overriding |
|---|---|
| The process of defining functions having same name with different parameter list is called Method Overloading | When the method in the subclass have same name and signature as Super class , then subclass method overrides super class method then it is called as Method Overriding. |
| It is a relationship between methods in same class. | It is a relationship between methods in superclass and subclass. |
| It is static binding- compile time | It is dynamic binding-runtime |
| It is a concept of compile time polymorphism | It is a concept of runtime polymorphism. |
| Return type do not affect method overloading | Return types , method name and signature of overridden method should be identical. |

NOTE:

if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

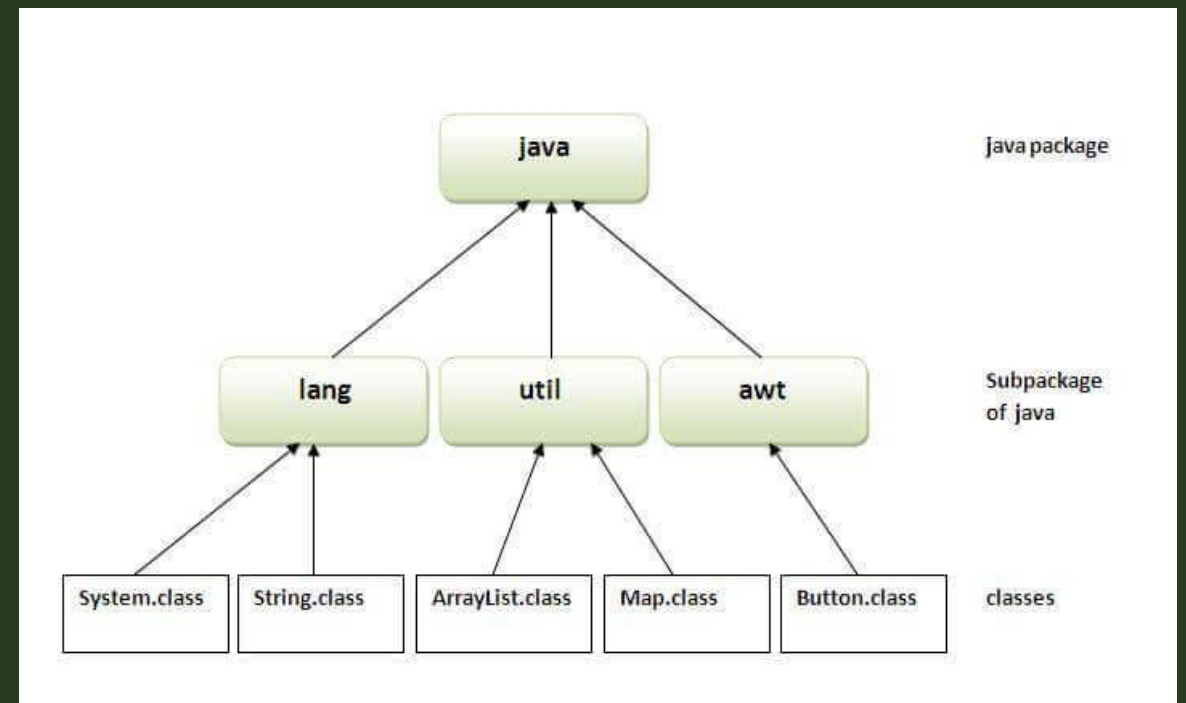
At run-time, it depends on the type of the object being referred to (not the type of the reference variable) determines which version of an overridden method will be executed .



Package

Introduction

- Package in Java is a mechanism to encapsulate a group of similar types of classes ,sub packages and interfaces.
- Package names and directory structure are closely related.



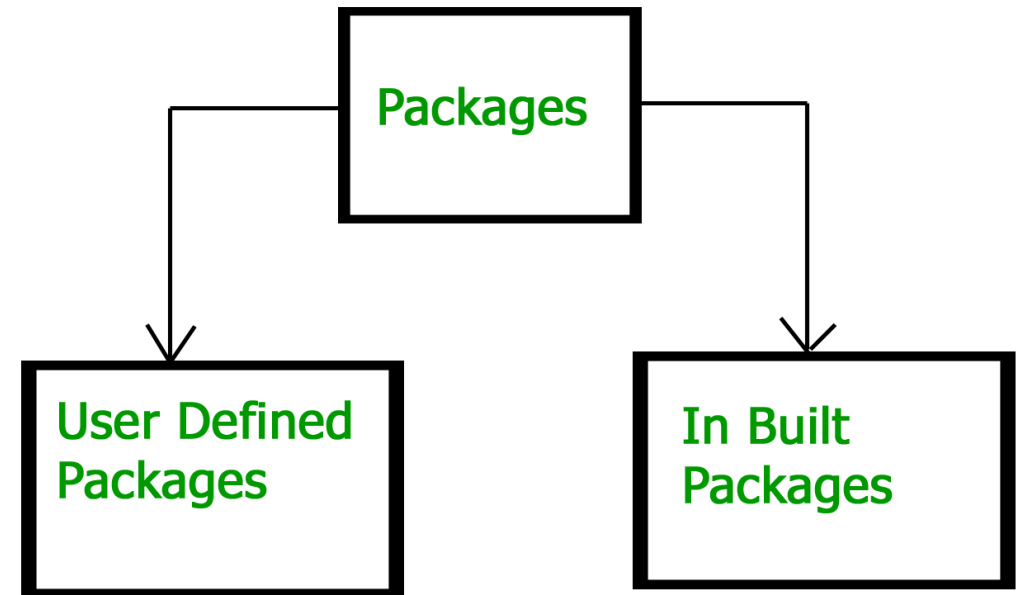
Types of Packages

- **Built-in Packages**

- These packages consist of many built-in classes which are a part of Java **API**.
- Example:
 - java.lang
 - java.util
 - java.io
 - Java.net

- **User defined Packages**

- These are the packages that are defined by the user.





Why?

- It prevent name conflicts related to classes.
- Makes locating and usage of classes and interfaces easier.
- Provides controlled access .

Defining a User Defined Package

- Include **Package** command as first statement in a Java source file.
- Any class declared within that source file will belong to specified package
- Package statement define namespace in which classes are stored.

- 
- if you omit Package statement then classes are put into default package, which has no name.
 - default package is fine for short programs , but it is not appropriate for real time applications.
- 

- The general form for Package statement

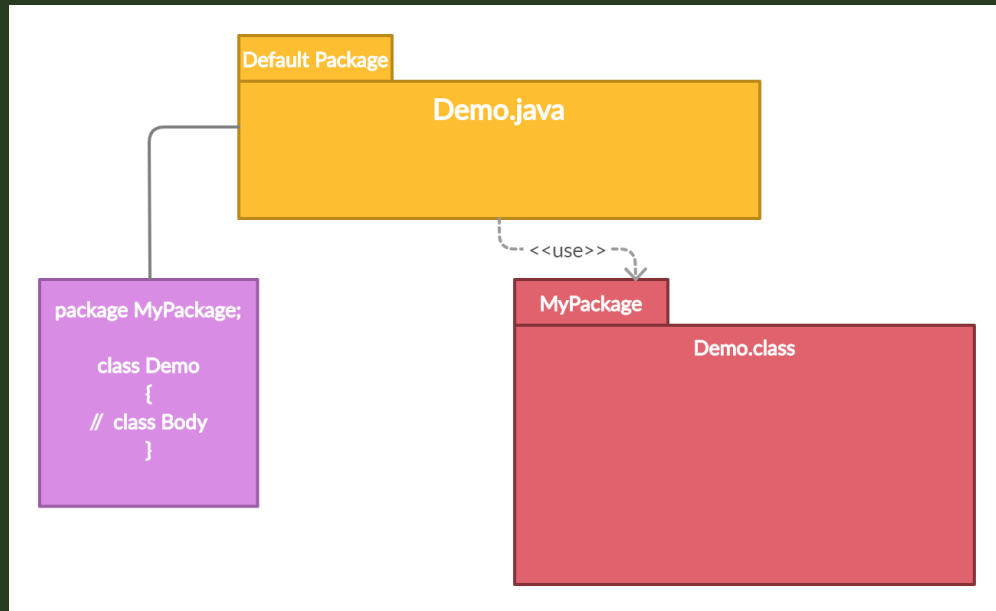
- **package** pkg;

- here pkg is the name of package.

- **Example:**

- **package** Mypackage;

- above statement creates package called **Mypackage**



Java uses file system directories to store packages.



the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.



The **package** statement simply specifies to which package the classes defined in a file belong.

Steps to Create User Defined Function

- Create a Java Source file with package statement along with Package name.

- Example: **A.java**

```
package mypackagedemo;  
class A  
{  
    public static void main(String[] args)  
    {  
        System.out.println("This is my first PackageDemo");  
    }  
}
```

- Compile java source code file along with **-d** option to generate appropriate directory structure for defined package.

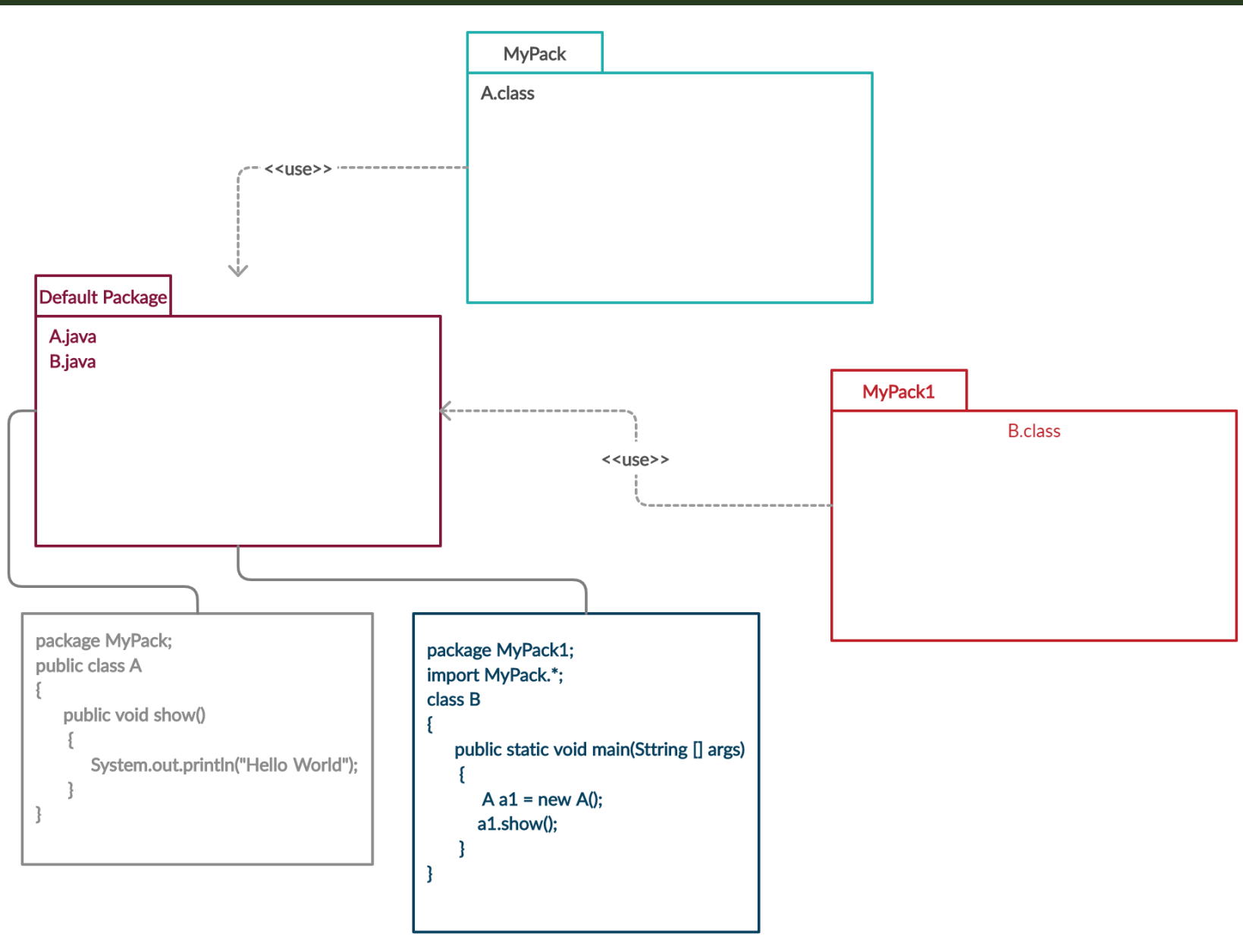
- Syntax:
 - `javac -d <directory_path> <javafilename>`
- Example
 - `javac -d . A.java`

- Run program from **immediate parent directory of the specified package name** along with **fully qualified name** of your **.class** file

- Syntax:
 - `java <packagename>.<classfile_name>`
- Example:
 - `java mypackagedemo.A`

How to Access classes defined in one package from other package?

- `import <packagename>.*;`
- `import`
 `<packagename>.classname;`
- fully qualified name



Access Control in JAVA

- **Access Control** is the mechanism by which you can precisely control access to the various members of a class.
- In Java how a member can be accessed is determined by the **access specifier**.
- Java supports reach set of access specifiers like
 - public
 - private
 - protected
 - default

Public

- When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code.
- The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package

Private

- When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.
- The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

Protected

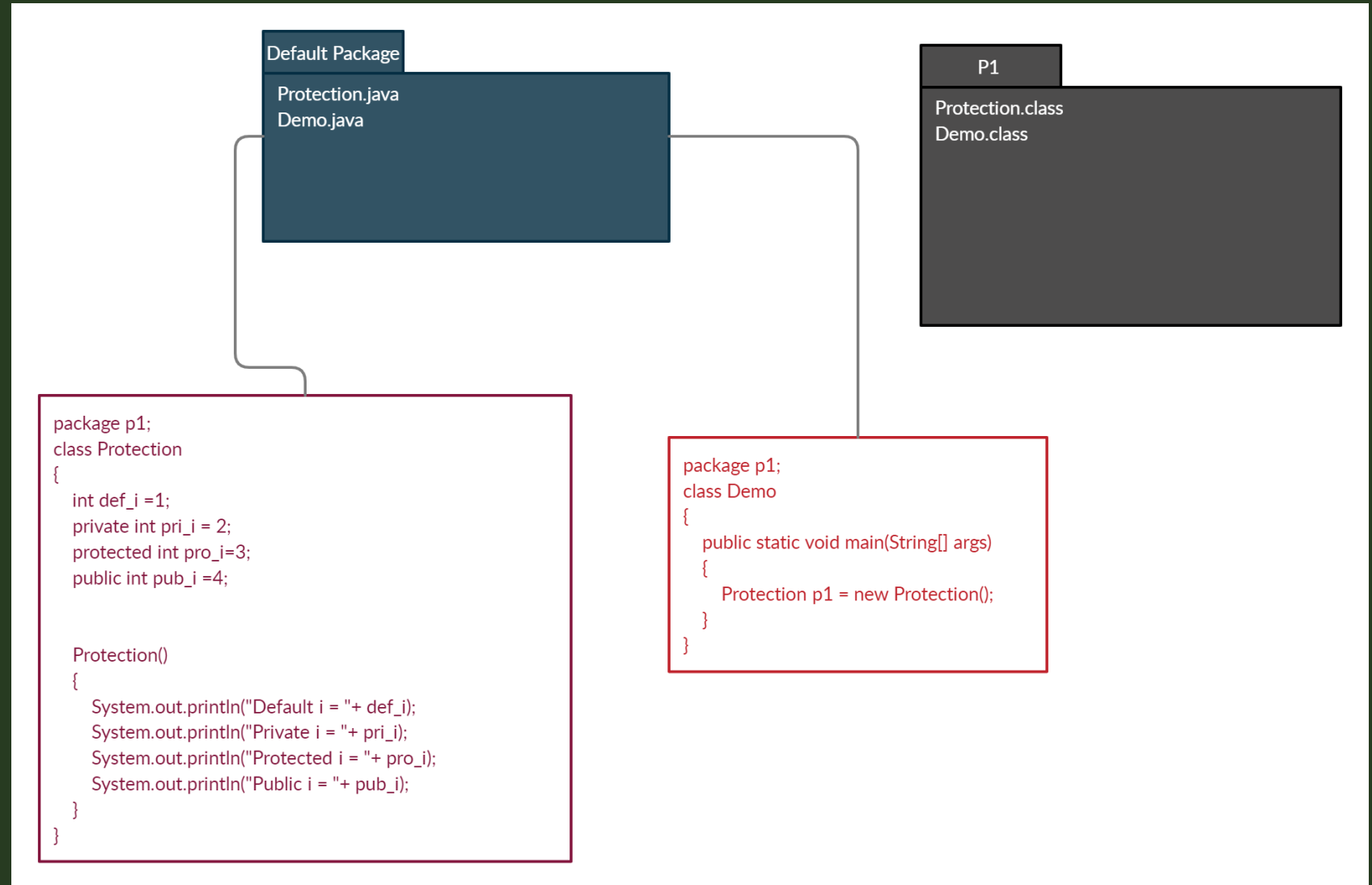
- The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
- The **protected access modifier** is accessible within package and outside the package but through inheritance only.
- It can't be applied on the class.

Default

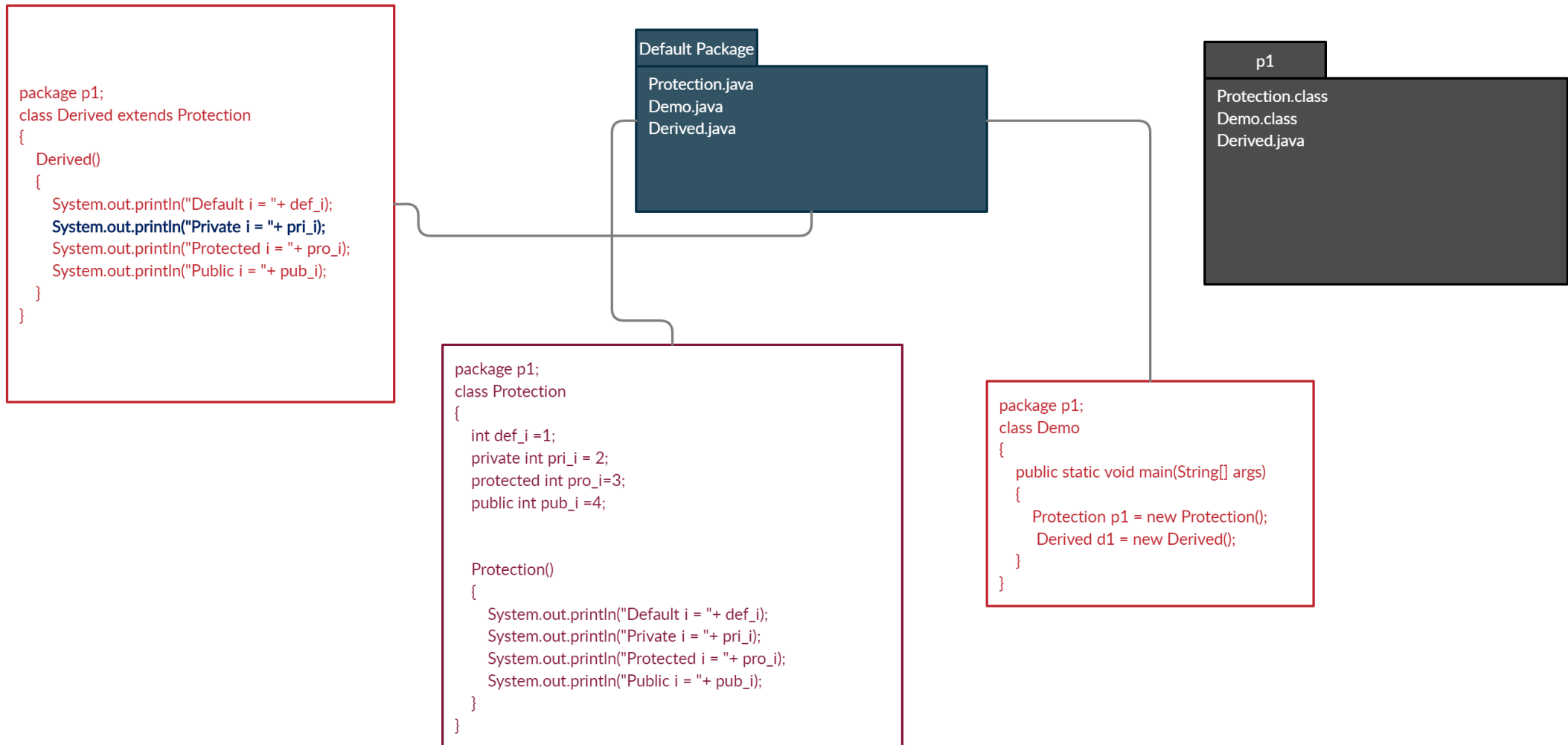
- The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

| | default | private | protected | public |
|--------------------------------|---------|---------|-----------|--------|
| Same Class | Yes | Yes | Yes | Yes |
| Same package subclass | Yes | No | Yes | Yes |
| Same package non-subclass | Yes | No | Yes | Yes |
| Different package subclass | No | No | Yes | Yes |
| Different package non-subclass | No | No | No | Yes |

Same Class



Same Package subclass





Same Package Non subclass




```
package p1;
class Protection
{
    int def_i =1;
    private int pri_i = 2;
    protected int pro_i=3;
    public int pub_i =4;

    Protection()
    {
        System.out.println("Default i = "+ def_i);
        System.out.println("Private i = "+ pri_i);
        System.out.println("Protected i = "+ pro_i);
        System.out.println("Public i = "+ pub_i);
    }
}
```

```
package p1;
class Derived extends Protection
{
    Derived()
    {
        System.out.println("Default i = "+ def_i);
        System.out.println("Private i = "+ pri_i);
        System.out.println("Protected i = "+ pro_i);
        System.out.println("Public i = "+ pub_i);
    }
}
```

```
package p1;
class NonDerived
{
    NonDerived()
    {
        System.out.println("Default i = "+ def_i);
        System.out.println("Private i = "+ pri_i);
        System.out.println("Protected i = "+ pro_i);
        System.out.println("Public i = "+ pub_i);
    }
}
```

Default Package
Protection.java
Demo.java
Derived.java
NonDerived.java

```
package p1;
class Demo
{
    public static void main(String[] args)
    {
        Protection p1 = new Protection();
        Derived d1 = new Derived();
        NonDerived nd1 = new NonDerived() ;
    }
}
```

p1
Protection.class
Demo.class
Derived.class
NonDerived.class

Same Package Non subclass

Different Package subclass

```
package p1;
class Derived extends Protection
{
    Derived()
    {
        System.out.println("Default i = "+ def_i);
        System.out.println("Private i = "+ pri_i);
        System.out.println("Protected i = "+ pro_i);
        System.out.println("Public i = "+ pub_i);
    }
}
```

```
package p1;
class NonDerived
{
    NonDerived()
    {
        System.out.println("Default i = "+ def_i);
        System.out.println("Private i = "+ pri_i);
        System.out.println("Protected i = "+ pro_i);
        System.out.println("Public i = "+ pub_i);
    }
}
```

```
package p1;
class Demo
{
    public static void main(String[] args)
    {
        Protection p1 = new Protection();
        Derived d1 = new Derived();
        NonDerived nd1 = new NonDerived() ;
    }
}
```

```
package p1;
class Protection
{
    int def_i =1;
    private int pri_i = 2;
    protected int pro_i=3;
    public int pub_i =4;

    Protection()
    {
        System.out.println("Default i = "+ def_i);
        System.out.println("Private i = "+ pri_i);
        System.out.println("Protected i = "+ pro_i);
        System.out.println("Public i = "+ pub_i);
    }
}
```

Default Package

- Protection.java
- Demo.java
- Derived.java
- NonDerived.java
- otherDemo.java
- otherDerived.java

p2

- otherDemo.class
- otherDerived.class

```
package p2;
import p1.*;
class otherDerived extends ProtectionA
{
    otherDerived()
    {
        System.out.println("otherDerived Constructor...");
        //System.out.println("Default i = "+ def_i);
        //System.out.println("Private i = "+ pri_i);
        System.out.println("Protected i = "+ pro_i);
        System.out.println("Public i = "+ pub_i);
    }
}
```

```
package p2;
class otherDemo
{
    public static void main(String[] args)
    {
        otherDerived od = new otherDerived();
    }
}
```

p1

- Protection.class
- Demo.class
- Derived.class
- NonDerived.class

Different Package non-subclass

```
package p1;
class Derived extends Protection
{
    Derived()
    {
        System.out.println("Default i = "+ def_i);
        System.out.println("Private i = "+ pri_i);
        System.out.println("Protected i = "+ pro_i);
        System.out.println("Public i = "+ pub_i);
    }
}
```

```
package p1;
class NonDerived
{
    NonDerived()
    {
        System.out.println("Default i = "+ def_i);
        System.out.println("Private i = "+ pri_i);
        System.out.println("Protected i = "+ pro_i);
        System.out.println("Public i = "+ pub_i);
    }
}
```

```
package p1;
class Demo
{
    public static void main(String[] args)
    {
        Protection p1 = new Protection();
        Derived d1 = new Derived();
        NonDerived nd1 = new NonDerived();
    }
}
```

```
package p1;
class Protection
{
    int def_i =1;
    private int pri_i = 2;
    protected int pro_i=3;
    public int pub_i =4;

    Protection()
    {
        System.out.println("Default i = "+ def_i);
        System.out.println("Private i = "+ pri_i);
        System.out.println("Protected i = "+ pro_i);
        System.out.println("Public i = "+ pub_i);
    }
}
```

p2

```
otherDemo.class
otherDerived.class
otherNonDerived.class
```

```
package p2;
import p1.*;
class otherDerived extends ProtectionA
{
    otherDerived()
    {
        System.out.println("otherDerived Constructor...");
        //System.out.println("Default i = "+ def_i);
        //System.out.println("Private i = "+ pri_i);
        System.out.println("Protected i = "+ pro_i);
        System.out.println("Public i = "+ pub_i);
    }
}
```

```
package p2;
class otherDemoA
{
    public static void main(String[] args)
    {
        otherDerivedA od = new otherDerivedA();
        otherNonDerivedA onda = new otherNonDerivedA();
    }
}
```

```
package p2;
import p1.*;
class otherNonDerivedA
{
    otherNonDerivedA()
    {
        ProtectionA p = new ProtectionA();
        System.out.println("otherDerived Constructor...");
        //System.out.println("Default i = "+ p.def_i);
        //System.out.println("Private i = "+ p.pri_i);
        //System.out.println("Protected i = "+ p.pro_i);
        System.out.println("Public i = "+ p.pub_i);
    }
}
```

Default Package

```
Protection.java
Demo.java
Derived.java
NonDerived.java

otherDemo.java
otherDerived.java
otherNonDerived.java
```

p1

```
Protection.class
Demo.class
Derived.class
NonDerived.class
```



INTERFACE IN JAVA

Introduction

- The interface in Java is *a mechanism to achieve abstraction*.
- Using interface you can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes , but they lack instance variable and their methods are declared without any body.
- Once an interface is defined any no of classes can implement it by inheriting abstract methods of the interface.
- it is a reference type in Java.

interface Remote



volumeDown()

volumeUp()

powerOn()

powerOff()

class Television





Why?

Full Abstraction

Multiple Inheritance

Defining an Interface

- The general form of an interface is:
- <interface_name> is the name of interface and it should be valid identifier name.
- name of source file in which interface resides should be same as **interface name**.

```
interface <interface_name> {  
    // declare constant fields  
    // declare abstract methods  
}
```


Characteristics



methods declared in interface is by default abstract .



variables declared inside interface is implicitly static and final.



All methods and variables are public implicitly.

Example:

```
interface interfaceA
{
    int a=1;
    void display();
}
```

Implementing an Interface

- Once an interface has been defined one or more class can implement that interface.
- To implement interface include **implements** clauses in class definition and then create methods defined by the interfaces.
- The general form of a class that implement interface is:

```
Class classname implements interface [,interface]  
{  
    // class-body  
}
```



- **if class implements more than one interface** , the interfaces are separated with a comma.
- The methods that implement an interface must be declared **public**.
- Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

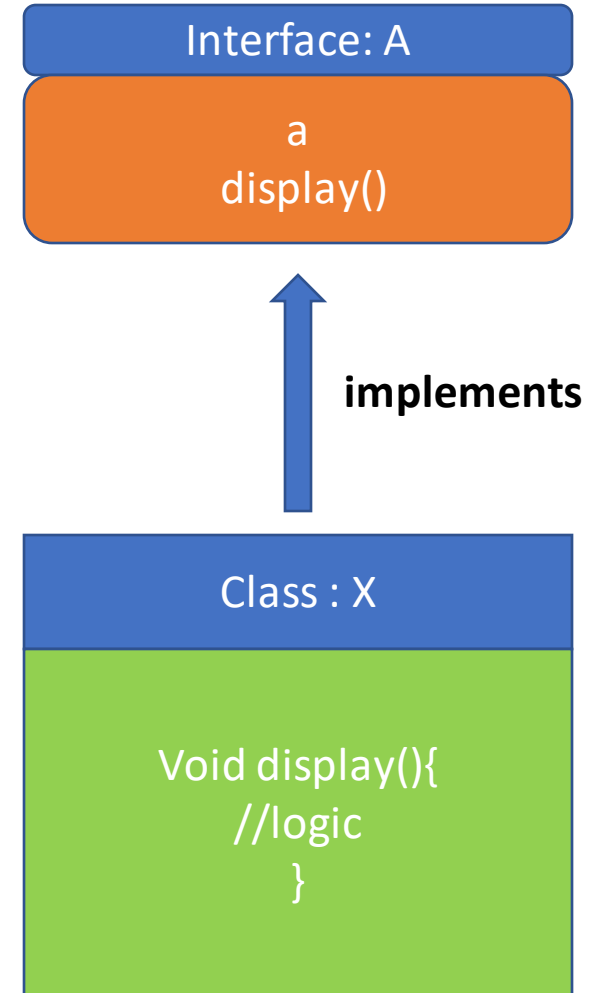
Example

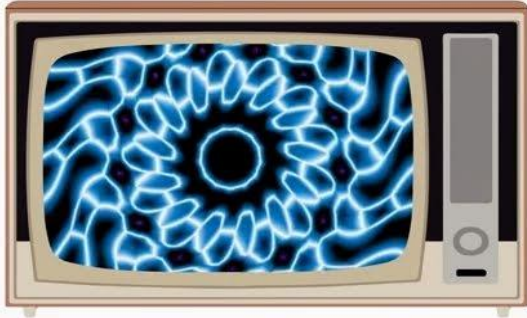
Interface A

```
{  
    int a =5;  
    void display();  
}
```

Class X implements A

```
{  
    void display()  
    {  
        System.out.println("InterfaceA = " + a);  
    }  
}
```





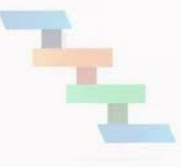
```
interface Remote {
    boolean powerSwitch();
    int decreaseVolume();
    int increaseVolume();
    double channelTuning(int channel);
}
```

```
abstract public class Remote {
    abstract public boolean powerSwitch();
    abstract public int increaseVolume();
    abstract public int decreaseVolume();
    abstract public double channelTuning(int channel);
}
```

```
class Application {
    public static void main(String args[]){
        Remote remoteTelevision =
            new Television(10.5,7,9);

        remoteTelevision.powerSwitch();
        remoteTelevision.decreaseVolume();
        remoteTelevision.increaseVolume();
        remoteTelevision.channelTuning(2);
    }
}
```

```
class Television extends Remote
    implements Remote{
    private double width
    private double height
    private double screenSize
    private int maxVolume
    private int volume
    private boolean power
    public Television(double width,
        double height,
        double screenSize){
        this.width=width;
        this.height=height;
        this.screenSize=screenSize;
    }
    public double channelTuning(int channel){
        switch(channel){
            case 1: return 34.56;
            case 2: return 54.89;
            case 3: return 73.89;
            case 4: return 94.98;
        } return 0;
    }
    public int decreaseVolume(){
        if(0<volume) volume--;
        return volume;
    }
    public void powerSwitch(){
        this.power = !power;
    }
    public int increaseVolume(){
        if(maxVolume>volume) volume++;
        return volume;
    }
}
```



InvolveInnovation

Similarity Between Class And Interface



An interface can contain any no 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.

Difference Between Class and Interface

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

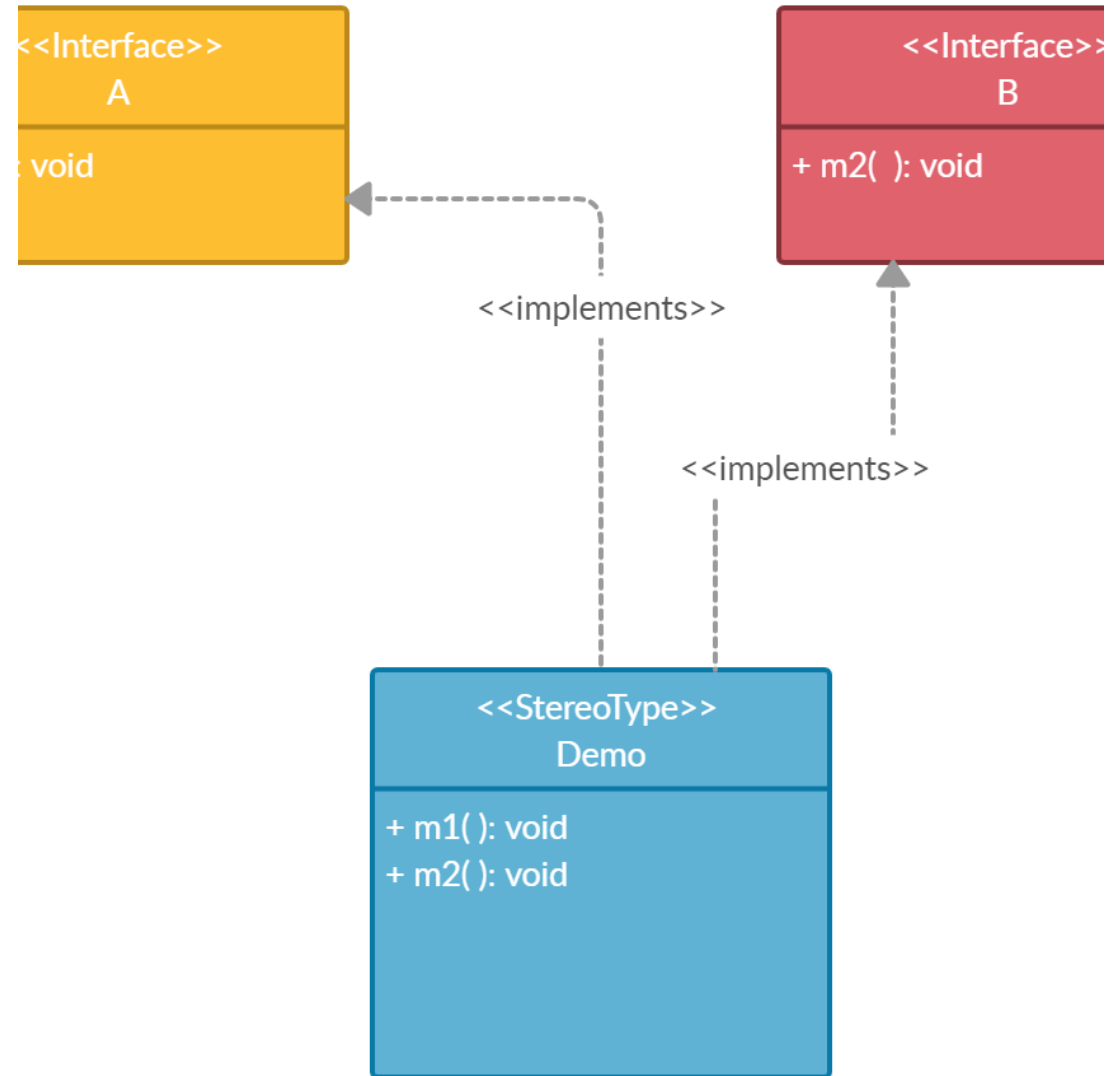
Abstract class v/s Interface

| Abstract class | Interface |
|---|---|
| Abstract class can have abstract and non-abstract methods. | Interface can have only abstract methods. |
| Abstract class doesn't support multiple inheritance . | Interface supports multiple inheritance . |
| The abstract keyword is used to declare abstract class. | The interface keyword is used to declare interface. |
| An abstract class can be extended using keyword "extends". | An interface can be implemented using keyword "implements". |
| A Java abstract class can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| Abstract class can provide the implementation of interface . | Interface can't provide the implementation of abstract class . |

Implementing Multiple interfaces in One Class

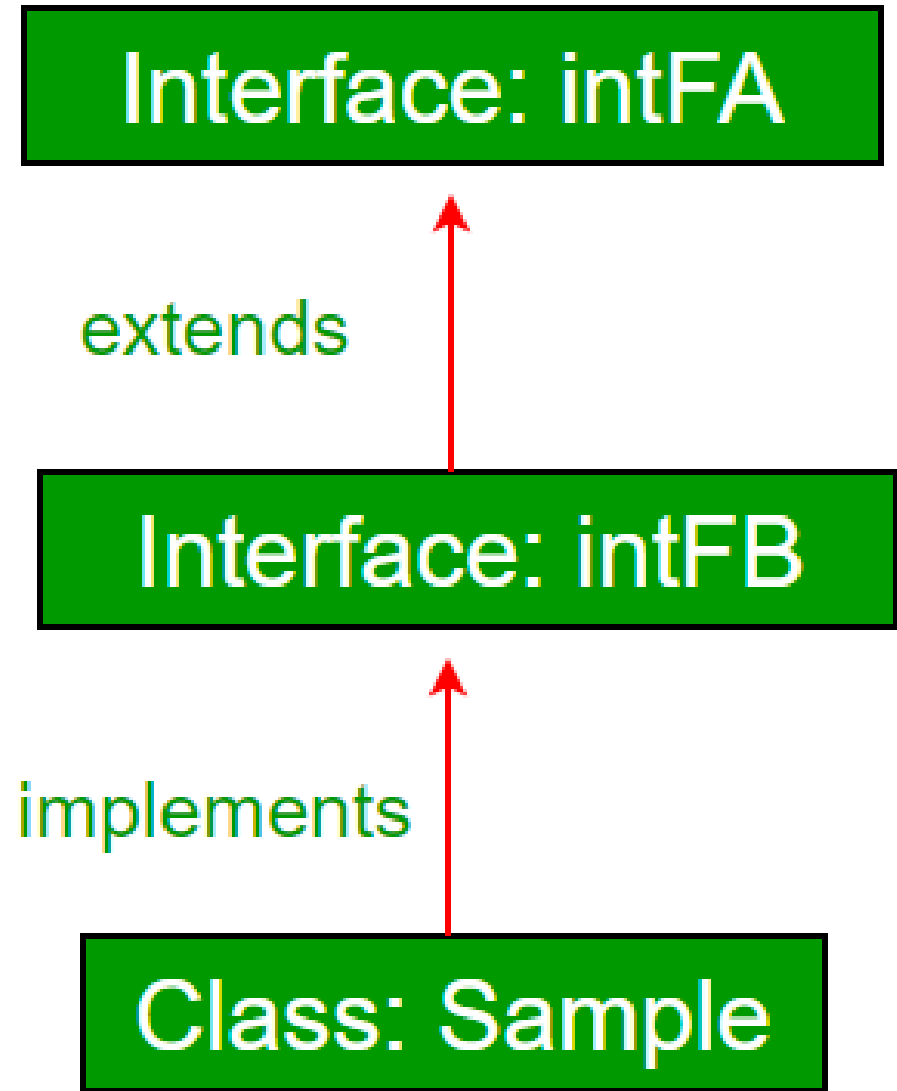
```
interface A
{
    void m1();
}
interface B
{
    void m2();
}
class Demo12 implements A,B
{
    void m1()
    {
        System.out.println("M1");
    }

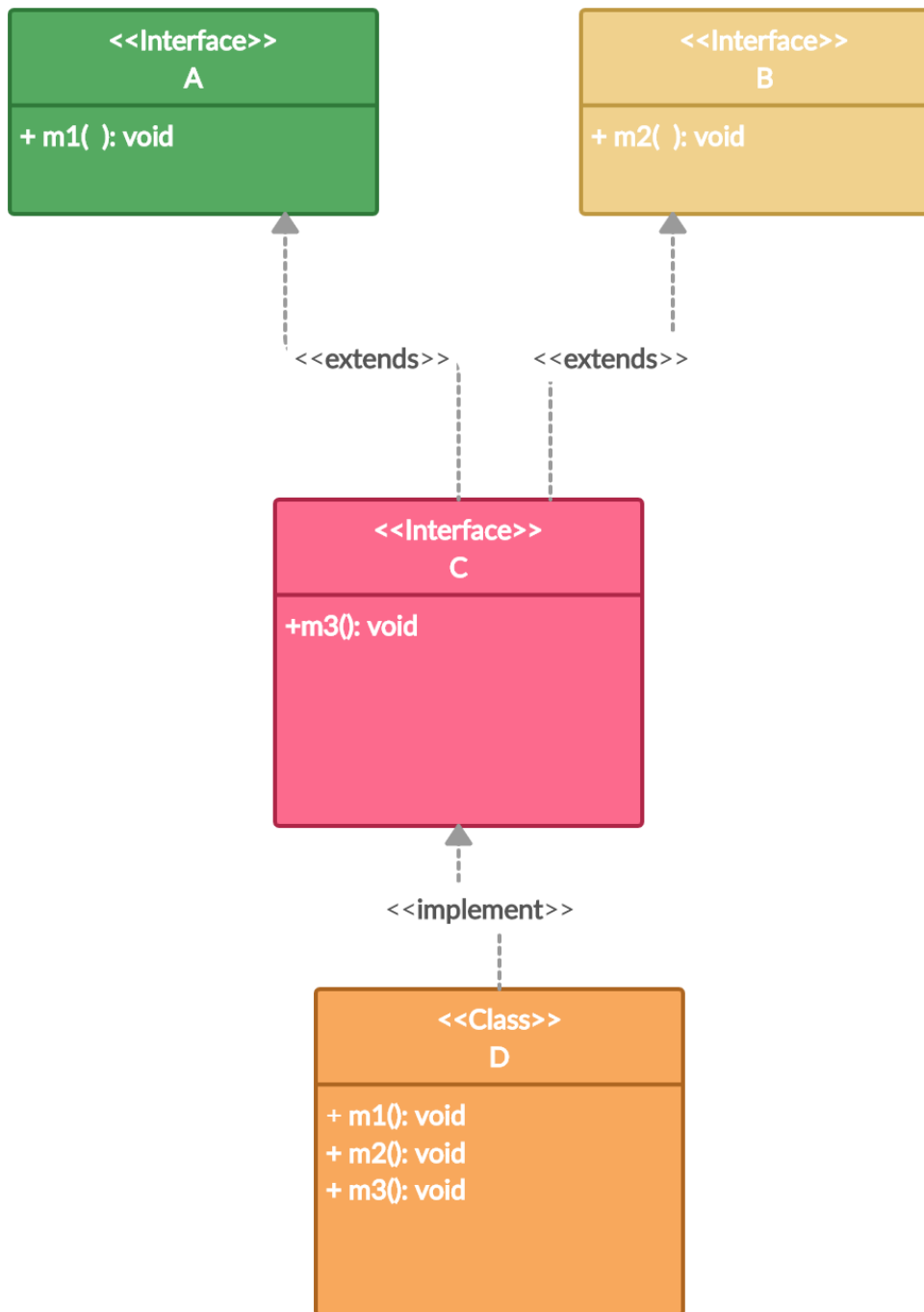
    void m2()
    {
        System.out.println("M2");
    }
}
```



Interface Inheritance

- one interface can inherit another interface using "extends" keyword.
- When a class implements an interface which inherits another interfaces, it must provide implementation for all methods defined within the interface inheritance chain.





Multiple Inheritance in Java

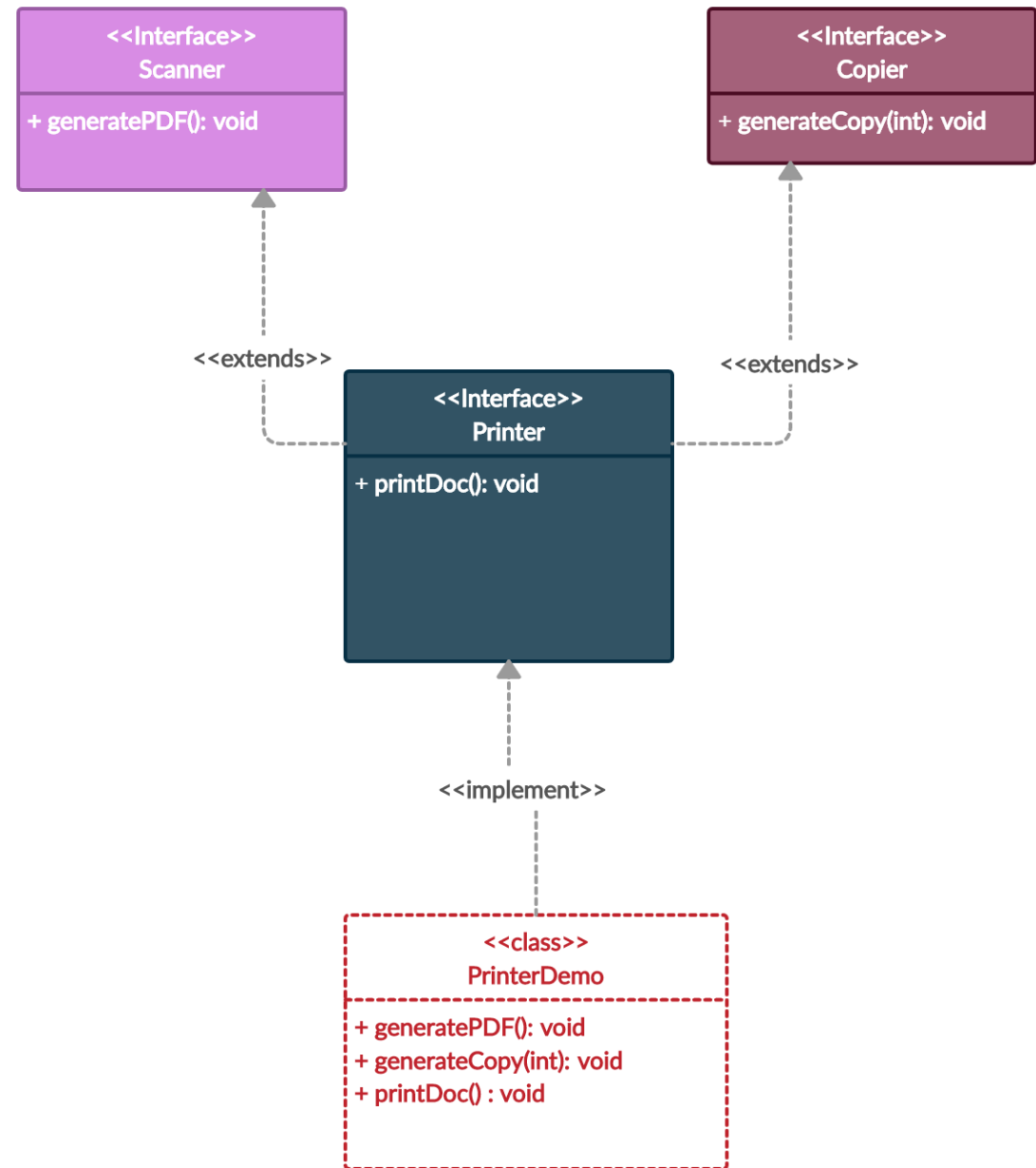
```
interface A
{
    void m1();
}
```

```
interface B
{
    void m2();
}
```

```
interface C extends A, B
{
    void m3();
}
```

```
class D implements C
{
    public void m1()
    {
        System.out.println("M1");
    }
    public void m2()
    {
        System.out.println("M2");
    }
    public void m3()
    {
        System.out.println("M3");
    }
}
```

Multiple Inheritance



The background is a top-down view of a modern office space. It features several light-colored office chairs with metal frames and casters arranged around a large, round, light-grey table. The floor is a light grey with a subtle texture. In the top-left corner, there are faint, white, wavy lines. In the bottom-right corner, there are faint, white, wavy lines and a solid blue line that curves upwards.

ABSTRACT KEYWORD IN JAVA

Situation

- + Sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- + Such a class determines the nature of the methods that the subclasses must implement.
- + One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.
- + You may have methods that must be overridden by the subclass in order for the subclass to have any meaning.

SHAPE
void area();

```
graph BT; Circle((CIRCLE)) --> Shape[SHAPE]; Triangle((TRIANGLE)) --> Shape; Rectangle[RECTANGLE] --> Shape;
```

The diagram illustrates a class hierarchy where three subclasses (CIRCLE, TRIANGLE, and RECTANGLE) inherit from a base class (SHAPE). The base class has a method 'void area()'. Each subclass also has an 'area()' method with its own implementation. The CIRCLE class is represented by a red circle, the TRIANGLE class by an orange triangle, and the RECTANGLE class by a green rectangle. Arrows point from each subclass to the base class, indicating inheritance.

CIRCLE

```
void area(){  
  A = 3.14*r*R  
}
```

TRIANGLE

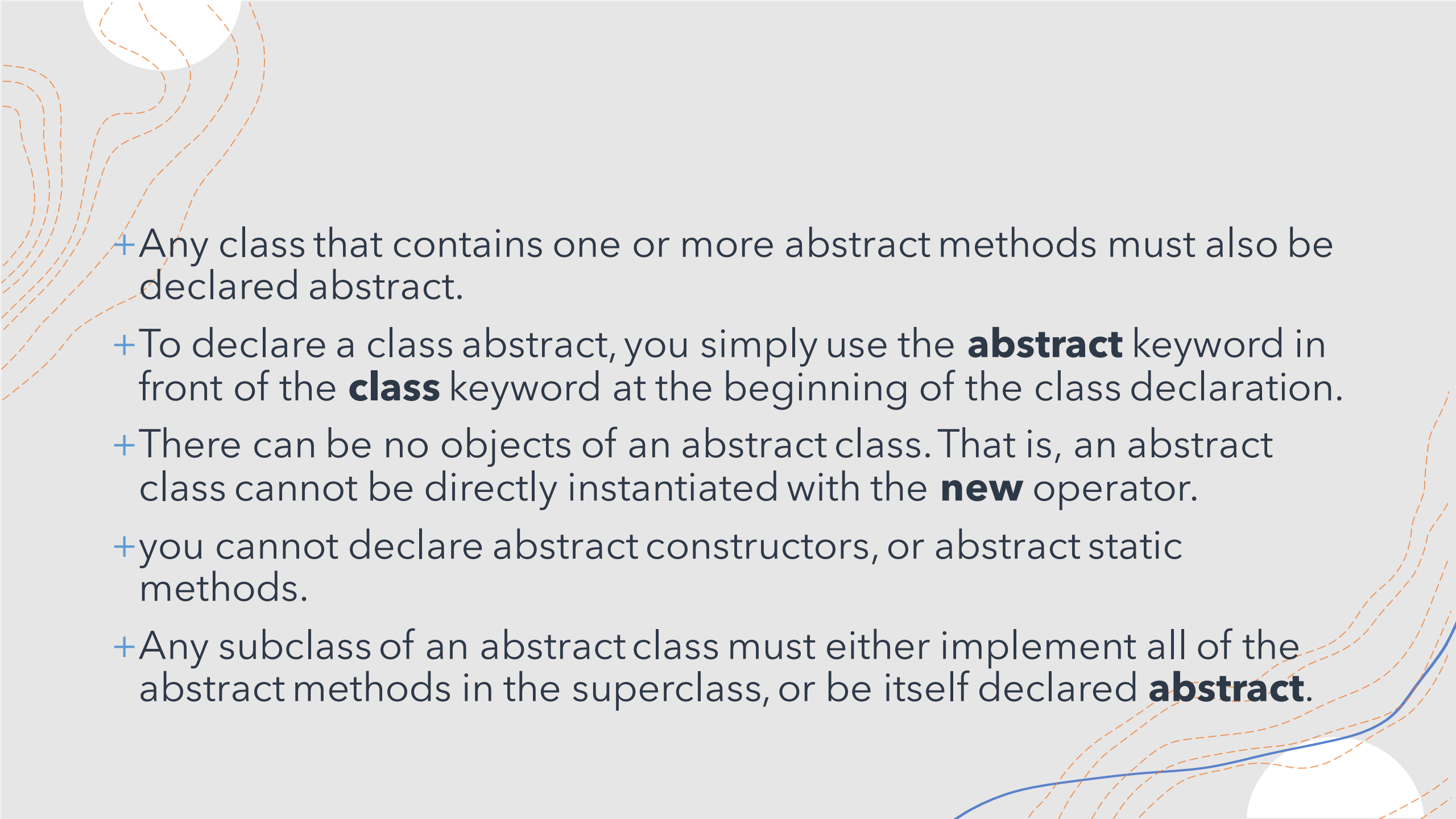
```
void area(){  
  A = (l*h)/2  
}
```

RECTANGLE

```
void area(){  
  A = (l*h)  
}
```


Abstract Method

- + A method without body (no implementation) is known as abstract method.
- + These methods are sometimes referred to as *subclass's responsibility* because they have no implementation specified in the superclass.
- + To declare an abstract method, use this general form:
+ **abstract** *type name*(*parameter-list*);

- 
- + Any class that contains one or more abstract methods must also be declared abstract.
 - + To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration.
 - + There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator.
 - + you cannot declare abstract constructors, or abstract static methods.
 - + Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract**.

Concrete Class v/s Abstract Class

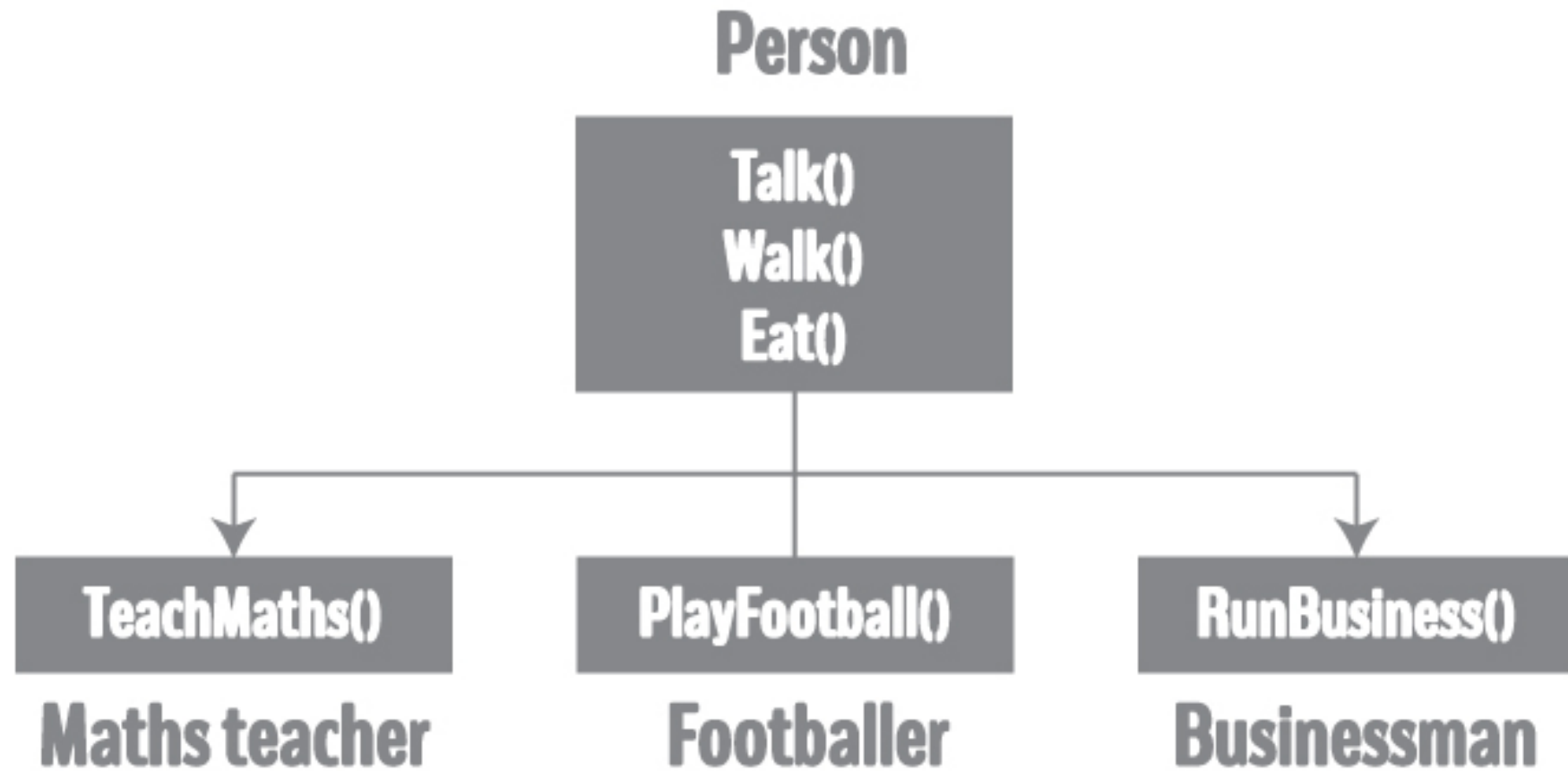
| Concrete Class | Abstract class |
|--|--|
| It provides complete representation of an object. | It provides partial representation of an object. |
| Concrete class always contain concrete methods. | Abstract class may or may not contain Concrete methods . |
| you can instantiate concrete class. | you cannot instantiate an abstract class |
| it is not mandatory to inherit concrete class to use it. | you need to inherit abstract class to use it. |



INHERITANCE

INTRODUCTION

- It is a mechanism in Java which allows one class to inherit features(variables ,methods) of another class .
 - Using the concept of inheritance we can create a general class , which contains common features of all the objects of that class. We can also derive another class from general class which inherits all the features of general class and have some specific features of its own.
 - it can be best understood in terms of Parent and Child relationship.
-



IMPORTANT TERMINOLOGY

- **Super Class:** The class whose features are inherited is known as super class(or a base class or a parent class).
 - **Sub Class:** The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the super class fields and methods.
-

-
- A subclass inherits variables and methods from its super class and all its ancestors but vice-versa is not possible
 - Subclass only inherits those super class members which are public and protected.
 - Super class can have any no of sub classes but sub class can directly have only one super class
 - Subclass can not inherits the constructor from its super class.
 - Subclass do not inherits a super class member if the subclass declares a member with same name.
 - in the case of member variables ,the member variable in the subclass hides the one in the super class.
 - in the case of member methods, the methods in subclass overrides the method in super class.
-

PURPOSE

Code Re-usability // reduce code redundancy

Polymorphism

TYPES OF INHERITANCE

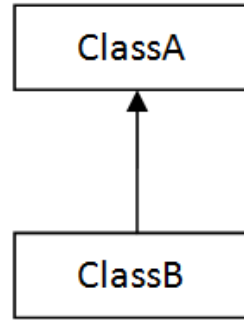
Single Level

Multilevel

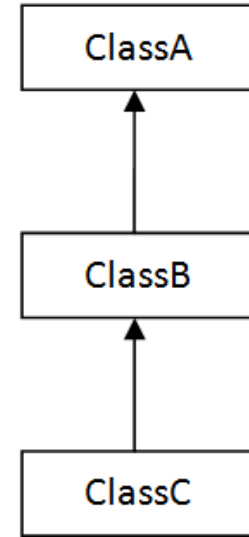
Hierarchical

NOTE :

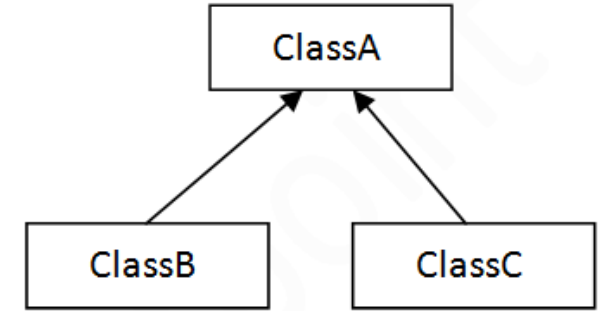
- **Multiple** inheritance is not supported in Java through class.
- Multiple and Hybrid inheritance is possible in java through **interfaces**.
- **No** class can be super class of itself.



1) Single



2) Multilevel



3) Hierarchical

EXTENDS KEYWORD

extends keyword is used to inherit a class.

Java allows one class to extend **only one** other class.

SYNTAX

class derived-class extends base-class

{

//methods and fields

}

Inheritance in Java

- **extends Keyword**
- **extends** is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

```
class Super {  
    ....  
    ....  
}  
class Sub extends Super {  
    ....  
    ....  
}
```

SINGLE LEVEL INHERITANCE

Types of Inheritance

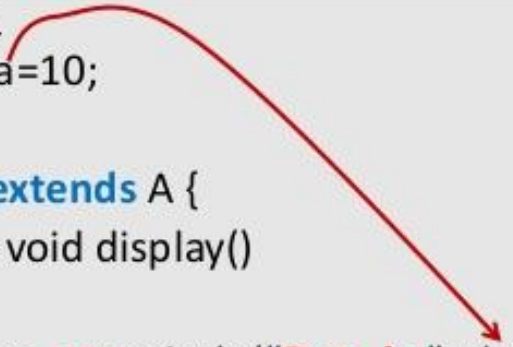
- **1) Single Inheritance**
- **Single inheritance** is easy to understand. When a class extends another one class only then we call it a single inheritance. The below flow diagram shows that class B extends only one class which is A. Here A is a **parent class** of B and B would be a **child class** of A.



Introduction to Java Programming Language

Single Inheritance Example

```
class A {  
    int data=10;  
}  
class B extends A {  
    public void display()  
    {  
        System.out.println("Data is:"+data);  
    }  
    public static void main(String args[])  
    {  
        B obj = new B();  
        obj.display();  
    }  
}
```

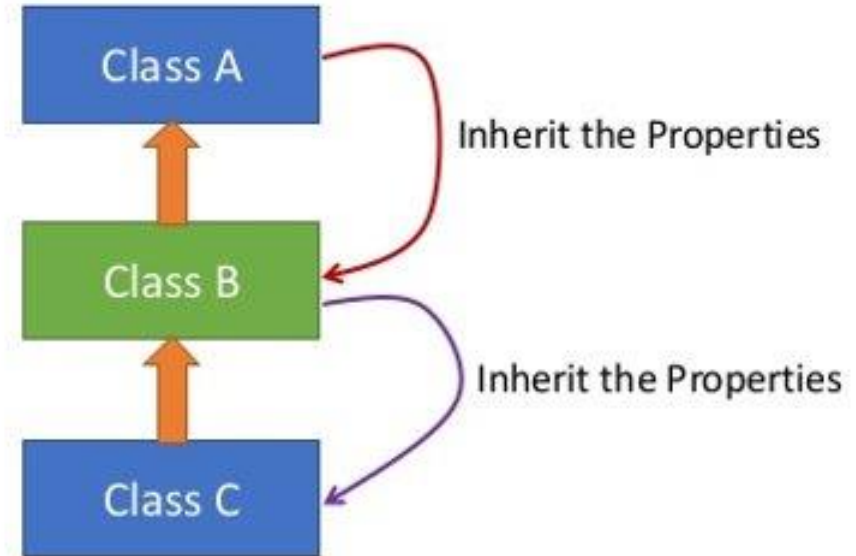


Child Class B
inherit/Access the
data field of Parent
Class A

Output is:
Data is:10

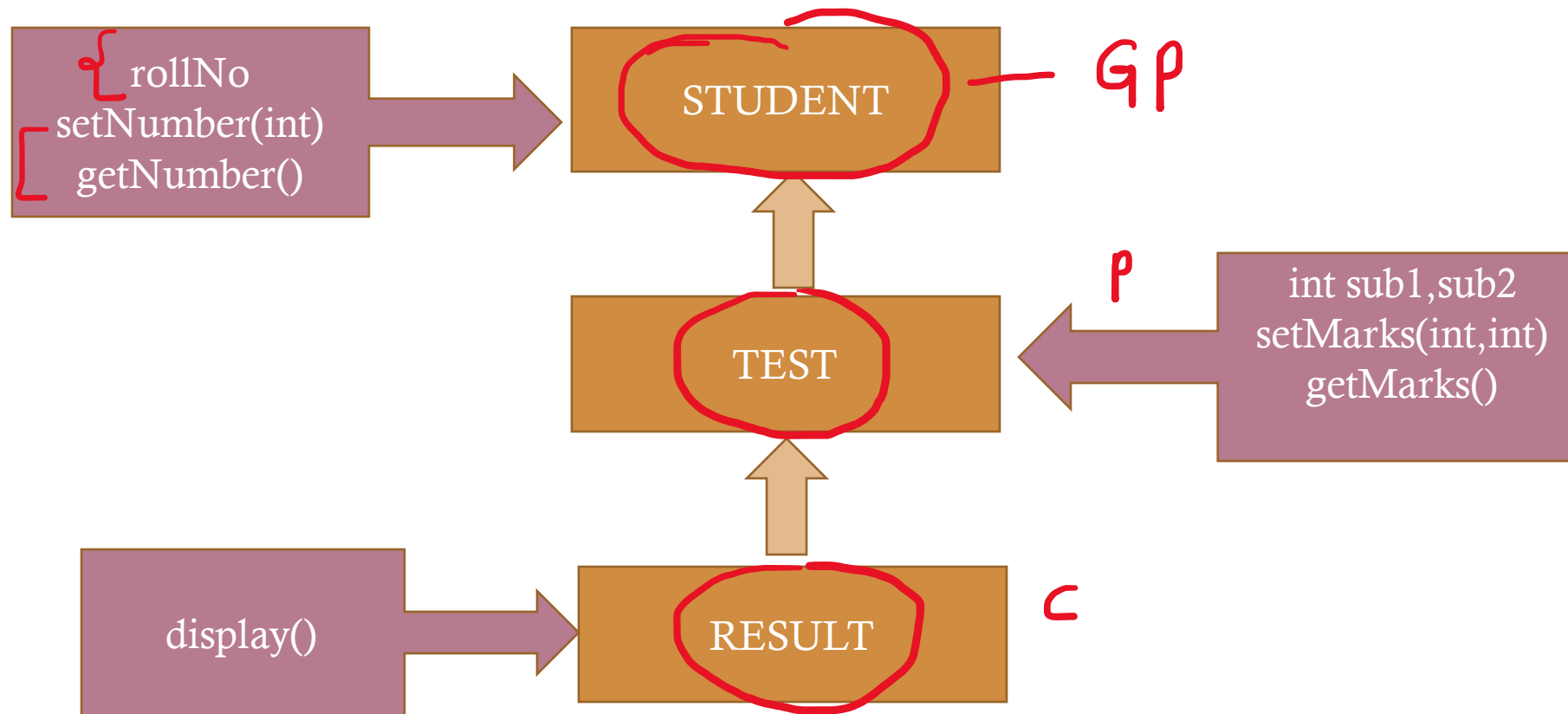
MULTILEVEL INHERITANCE

Multilevel Inheritance



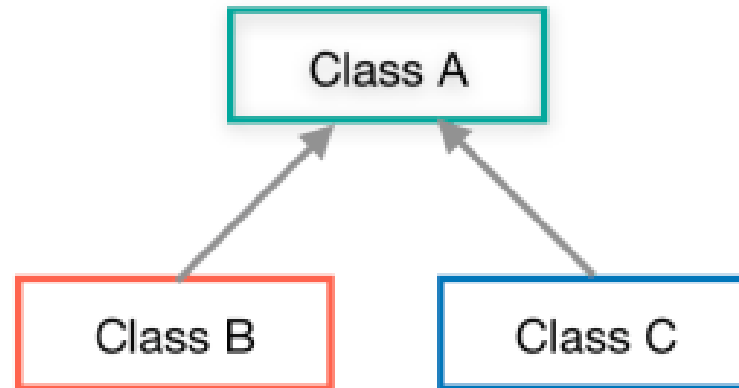
- Here class C inherits class B and class B inherits class A which means B is a parent class of C and A is a parent class of B. So in this case class C is implicitly inheriting the properties and method of class A along with B that's what is called multilevel inheritance.

EXAMPLE:



HIERARCHICAL

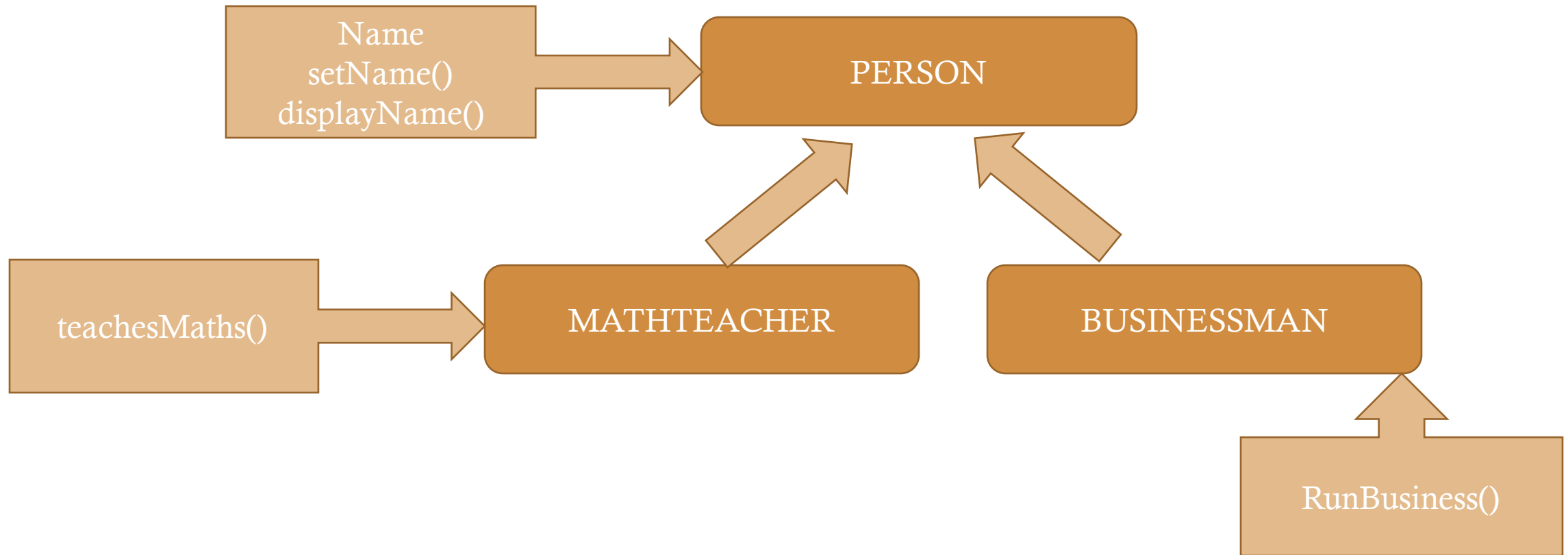
- In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.



Hierarchical
Inheritance

```
public class A {  
    .....  
}  
  
public class B extends A {  
    .....  
}  
  
public class C extends A {  
    .....  
}
```

EXAMPLE:





SUPER KEYWORD

Introduction

- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.
- Super can be used at
 - Variable level
 - Method level
 - Constructor level
- **super** has two general forms.
 - It is used to access a member of the superclass that has been hidden by a member of a subclass.
 - To calls the superclass' constructor or methods.

Access the Hidden data variable of the super class

- This form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass
- This usage has the following general form:
 - `super.member`
 - Here, *member* can be either a method or an instance variable.

TestMain.java

```
1 package Prashant;
2
3 class TestSuper {
4     int x = 120;
5 }
6
7 class TestSuper1 extends TestSuper {
8     int x = 180;
9
10    void display() {
11        System.out.println("Value of x variable " + super.x);
12    }
13 }
14
15 class TestMain {
16    public static void main(String[] args) {
17        TestSuper1 testSuper1 = new TestSuper1();
18        testSuper1.display();
19    }
20 }
21
```

When Constructors are called?

- In a class hierarchy constructors are called in the order of derivation ,from super class to subclass.

```
class P
{
    P()
    {
        System.out.println("P constructor");
    }
}
class Q extends P
{
    Q()
    {
        System.out.println("Q constructor");
    }
}
class R extends Q
{
    R()
    {
        System.out.println("R constructor");
    }
}
class CallDemo
{
    public static void main(String[] args)
    {
        R r = new R();
    }
}
```


Call Super class Constructors

- A subclass can call a constructor defined by its superclass by use of the following form of **super**:
 - `super(arg-list);`
 - Here, *arg-list* specifies any arguments needed by the constructor in the superclass.
- **super()** must always be the first statement executed inside a subclass' constructor.
- When a subclass calls **super()**, it is calling the constructor of its immediate superclass.

```
class A
{
    int i;
    int j;

    A(int x,int y)
    {
        System.out.println("In parameterized constructor of class A");
        i = x;
        j = y;
    }
}

class B extends A
{
    int k;

    B(int a,int b,int c)
    {
        super(a,b);
        System.out.println("In parameterized constructor of class B");
        k = c;
    }

    public void display()
    {
        System.out.println("i = "+i);
        System.out.println("j = "+j);
        System.out.println("k = "+k);
    }
}
```

