

Inheritance, Interface and Package.

Prepared By:
D R Gandhi

Learning Outcomes

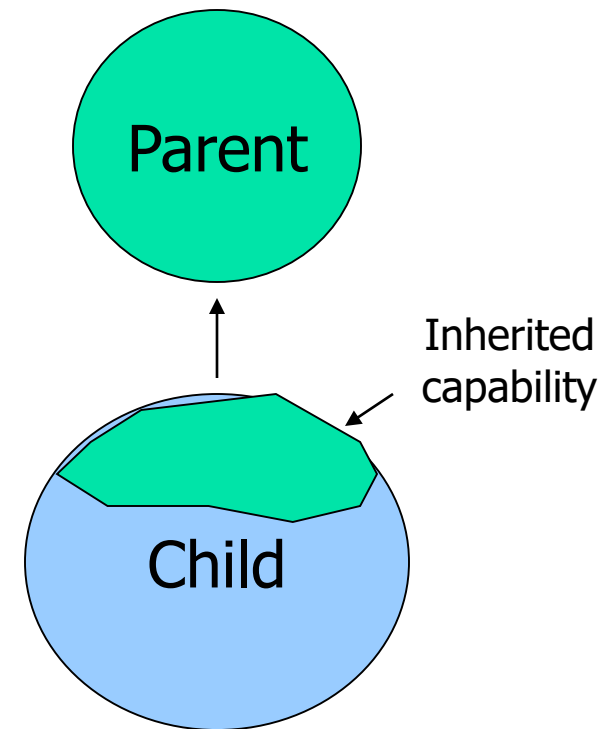
- Basics of Inheritance, Types of Inheritance, Method overriding, super and final keyword.
- Basics of Polymorphism, Types of Polymorphism, Difference between method overloading and method overriding.
- Dynamic method dispatch & Object class.
- Abstract classes v/s Interfaces, defining an interface, implementing interfaces, extending interfaces, default method, lambda expression.
- Creating package, setting a CLASSPATH, adding class and interfaces to a package, importing package, static import
- Java access modifier, Access and class hiding rules in a package.

Inheritance: Introduction

- Reusability--building new components by utilising existing components- is yet another important aspect of OO paradigm.
- It is always good/"productive" if we are able to reuse something that already exists rather than creating the same all over again.
- There are two ways to reuse existing classes:
 - Composition: defining a new class, which is composed of existing classes.
 - Inheritance : deriving a new class based on an existing class, with modifications or extensions.

Inheritance: Introduction

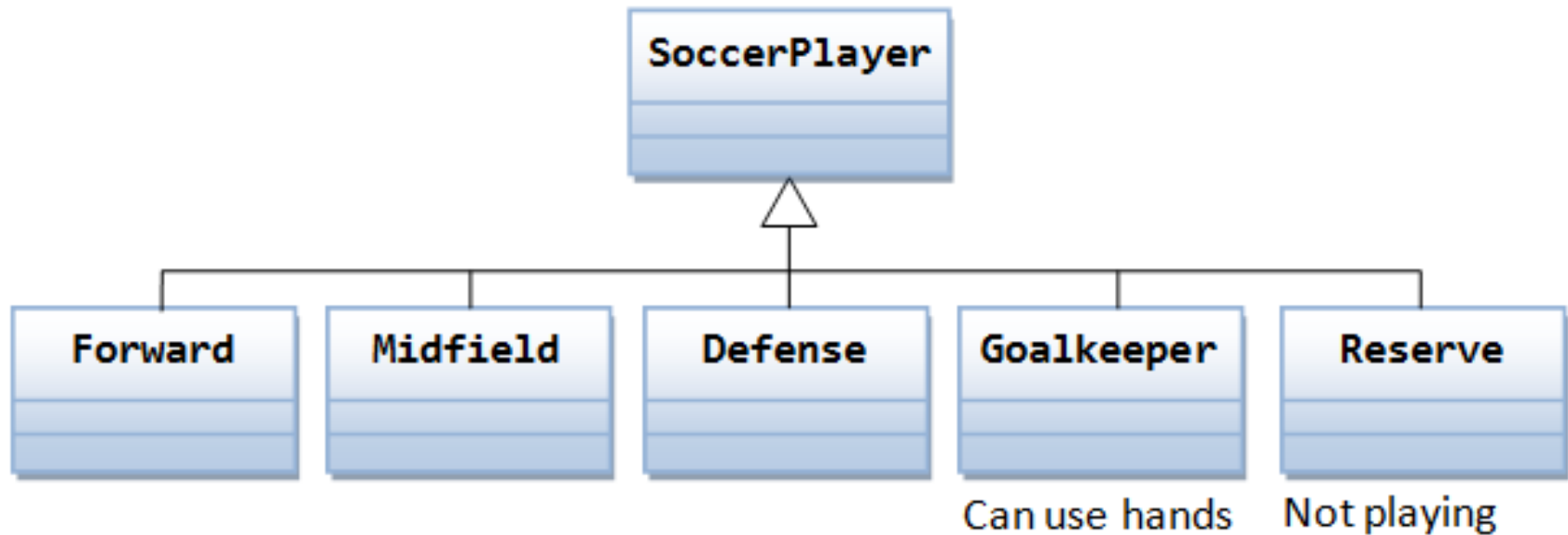
- This is achieved by creating new classes, reusing the properties of existing classes.
- This mechanism of deriving a new class from existing/old class is called "inheritance".
- The old class is known as "base" class, "super" class or "parent" class"; and the new class is known as "sub class", "derived class", "extended class" or "child class".



Inheritance: Introduction

- In OOP, to *avoid duplication and reduce redundancy* classes are organized in *hierarchy*.
- The classes in the lower hierarchy inherit all the variables (static attributes) and methods (dynamic behaviors) from the higher hierarchies.
- A class in the lower hierarchy is called a *subclass*.
- A class in the upper hierarchy is called a *superclass*.
 - By pulling out all the common variables and methods into the superclasses, and leave the specialized variables and methods in the subclasses, *redundancy* can be greatly reduced or eliminated as these common variables and methods do not need to be repeated in all the subclasses. For example,

Inheritance: Introduction



- A subclass inherits all the variables and methods from its superclasses, including its immediate parent as well as all the ancestors.
- subclass is not a "subset" of a superclass.
- subclass is a "superset" of a superclass.
- subclass inherits all the variables and methods of the superclass; in addition, it extends the superclass by providing more specialized variables and methods.

Defining a Sub class

- A subclass/child class is defined as follows:

```
class SubClassName extends SuperClassName
{
    fields declaration;
    methods declaration;
}
```

- The keyword “extends” signifies that the properties of super class are extended to the subclass.
- That means, subclass contains its own members as well of those of the super class.
- This kind of situation occurs when we want to enhance properties of existing class without actually modifying it.

Defining a Sub class

- A subclass/child class is defined as follows:

```
class Animal {  
    // methods and fields  
}  
  
// use of extends keyword to perform inheritance  
class Dog extends Animal {  
  
    // methods and fields of Animal  
    // methods and fields of Dog  
}
```

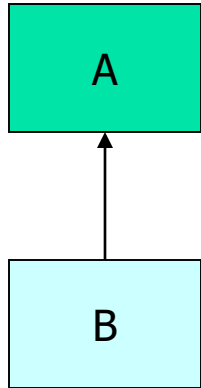

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

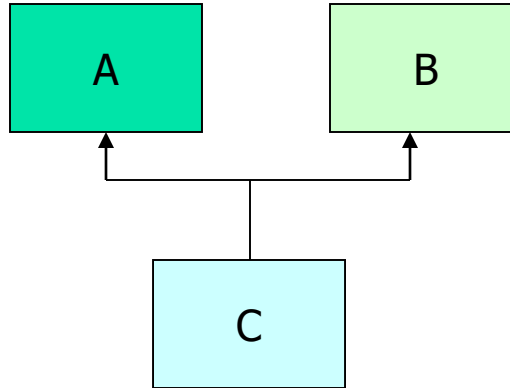
Types of Inheritance

- The inheritance allows subclasses to inherit all properties (variables and methods) of their parent classes. The different forms of inheritance are:
 - Single inheritance (only one super class)
 - Multiple inheritance (several super classes)
 - Hierarchical inheritance (one super class, many sub classes)
 - Multi-Level inheritance (derived from a derived class)
 - Hybrid inheritance (more than two types)
 - Multi-path inheritance (inheritance of some properties from two sources).

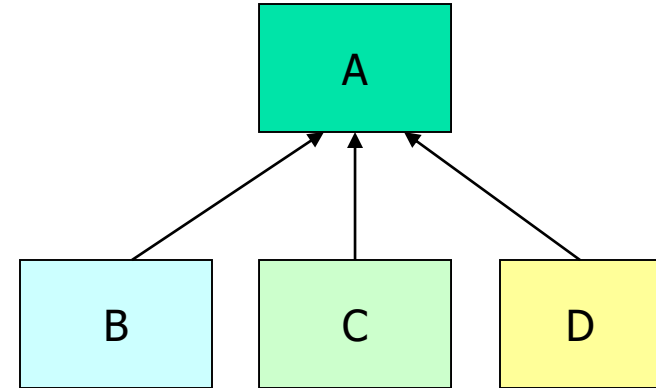
Types of Inheritance



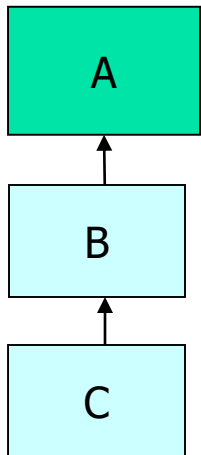
(a) Single Inheritance



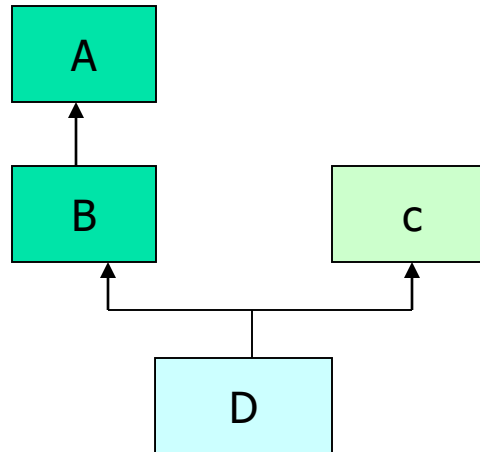
(b) Multiple Inheritance



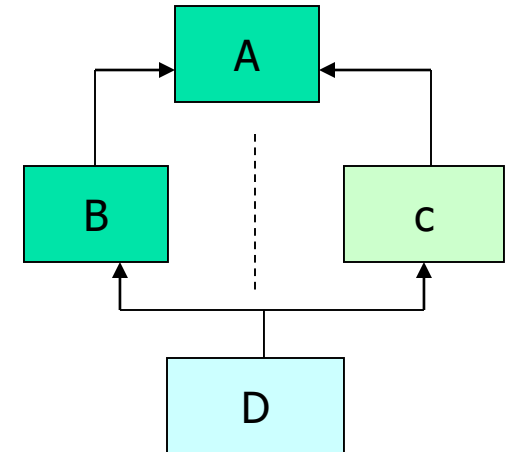
(c) Hierarchical Inheritance



(a) Multi-Level Inheritance



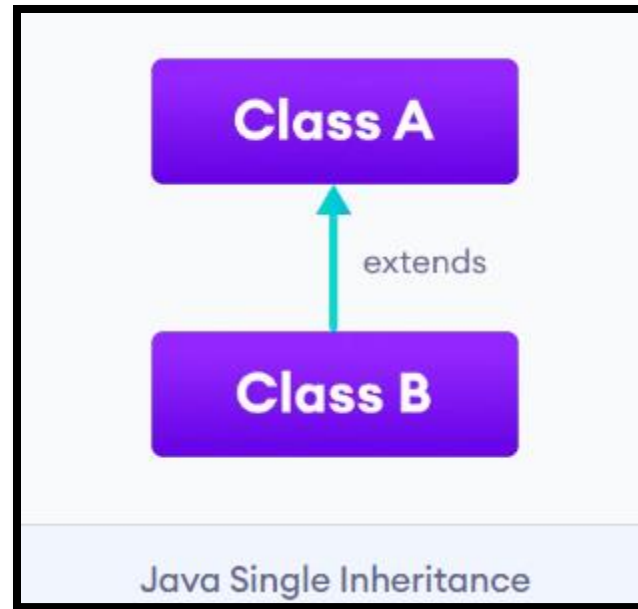
(b) Hybrid Inheritance



(b) Multipath Inheritance

Types of Inheritance

- **Single Inheritance:-**
- If a class is derived from a single class then it is called single inheritance.
- Class B is derived from class A.



Types of Inheritance

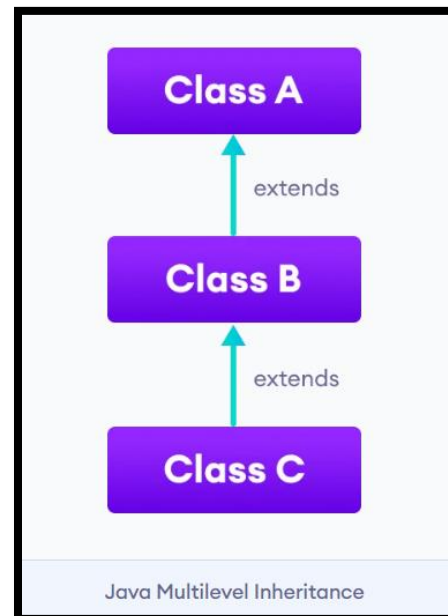
```
class Animal{  
void eat(){System.out.println(x:"eating...");}  
}  
class Dog extends Animal{  
void bark(){System.out.println(x:"barking...");}  
}  
public class singleinheritance{  
Run | Debug  
public static void main(String args[]){  
Dog d=new Dog();  
d.bark();  
d.eat();  
}} |
```

barking...
eating...

Types of Inheritance

■ Multilevel Inheritance:-

- A class is derived from a class which is derived from another class then it is called multilevel inheritance
- Here, class C is derived from class B and class B is derived from class A, so it is called multilevel inheritance.
- When there is a chain of inheritance, it is known as *multilevel inheritance*.



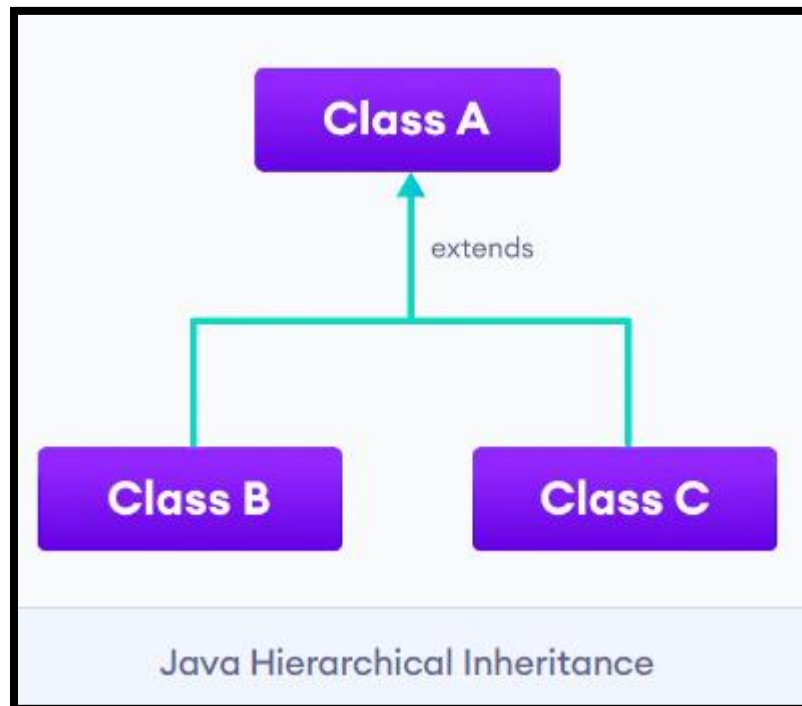
Types of Inheritance

```
class Animal{
    void eat(){System.out.println(x:"eating...");}
}
class Dog extends Animal{
    void bark(){System.out.println(x:"barking...");}
}
class BabyDog extends Dog{
    void weep(){System.out.println(x:"weeping...");}
}
public class multipleinheritance{
    Run | Debug
    public static void main(String args[]){
        BabyDog d=new BabyDog();
        d.weep();
        d.bark();
        d.eat();
    }
}
```

weeping...
barking...
eating...

Types of Inheritance

- **Hierarchical Inheritance:-**
- If one or more classes are derived from one class then it is called hierarchical inheritance.
- Here, class B and class C are derived from class A.



Types of Inheritance

```
//superclass
class Animal{
    void eat(){System.out.println(x:"eating...");}
}

//subclass 1
class Dog extends Animal{
    void bark(){System.out.println(x:"barking...");}
}

//subclass 2
class Cat extends Animal{
    void meow(){System.out.println(x:"meowing...");}
}

public class heirarchicalinheritance{
    Run | Debug
    public static void main(String args[]){
        Cat c=new Cat();
        c.meow(); //defined in cat class
        c.eat(); //inherited from animal class
        //c.bark();//C.T.Error
        Dog d=new Dog();
        d.bark();
        d.eat();
        //d.meow();//C.T.Error
    }}
}
```

meowing...
eating...
barking...
eating...

Types of Inheritance

- **Hybrid Inheritance:-**

- In general, the meaning of hybrid (mixture) is made of more than one thing. I
- In Java, the hybrid inheritance is the composition of two or more types of inheritance.
- The main purpose of using hybrid inheritance is to modularize the code into well-defined classes. It also provides the code reusability.
- Single and Multiple Inheritance (not supported but can be achieved through interface)
- Multilevel and Hierarchical Inheritance
- Hierarchical and Single Inheritance
- Multiple and Multilevel Inheritance

ance

```
class Animal {
public void eat()
{ System.out.println(x:"Eating...");
} }

class Dog extends Animal {
public void bark()
{ System.out.println(x:"barking");
} }

class Cat extends Animal {
public void meow()
{ System.out.println(x:"meowing");
} }

class bulldog extends Dog {
public void guard()
{ System.out.println(x:"guarding");
}}

public class hybrid{
Run | Debug
public static void main(String args[])
{ bulldog obj = new bulldog();
obj.guard();
obj.eat();
obj.bark();
} }
```

Guarding...
Eating...
Barking..

Types of Inheritance

■ **Multiple Inheritance:-**

- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
- Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes.
- If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.
- To remove ambiguity among subclasses.
- To provide more maintainable and clear design between classes.
- We use interface to achieve multiple inheritance.

Types of Inheritance

```
class A{
void msg(){System.out.println("Hello");}
}

class B{
void msg(){System.out.println("Welcome");}
}

class C extends A,B{//suppose if it were

public static void main(String args[]){
    C obj=new C();
    obj.msg();//Now which msg() method would be invoked?
}
}
```

Compile Time Error

Member Access and Inheritance

- *A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.*

Member Access and Inheritance

/* In a class hierarchy, private members remain private to their class.

This program contains an error and will not compile. */

// Create a superclass.

```
class A {  
    int i; // public by default  
    private int j; // private to A  
    void setij(int x, int y) {  
        i = x;  
        j = y;  
    }  
}
```

// A's j is not accessible here.

```
class B extends A {  
    int total;  
    void sum() {  
        total = i + j; // ERROR, j is not accessible here  
    }  
}
```

Extending Box class

/* programe defines three constructors to initialize the dimensions of a box various ways using constructor overloading. */

```
class Box {  
    double width;  
    double height;  
    double depth;  
    // constructor used when all dimensions specified  
    Box(double width, double height, double depth) {  
        this.width = width;  
        this.height = height;  
        this.depth = depth;  
    }  
    Box() {                // constructor used when no dimensions specified  
        width = -1; // use -1 to indicate  
        height = -1; // an uninitialized  
        depth = -1; // box  
    }  
    Box(double len) {      // constructor used when cube is created  
        width = height = depth = len;  
    }  
    double volume() {      // compute and return volume  
        return width * height * depth;  
    }  
}
```


Extending Box class

// Here, Box is extended to include weight.

```
class BoxWeight extends Box {  
    double weight; // weight of box  
    // constructor for BoxWeight  
    BoxWeight(double w, double h, double d, double m) {  
        width = w;  
        height = h;  
        depth = d;  
        weight = m;  
    }  
}
```

// Here, Box is extended to include color.

```
class ColorBox extends Box {  
    int color; // color of box  
    ColorBox(double w, double h, double d, int c) {  
        width = w;  
        height = h;  
        depth = d;  
        color = c;  
    }  
}
```

Extending Box class

```
class DemoBoxWeight {  
    public static void main(String args[]) {  
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);  
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);  
        double vol;  
        vol = mybox1.volume();  
        System.out.println("Volume of mybox1 is " + vol);  
        System.out.println("Weight of mybox1 is " + mybox1.weight);  
        System.out.println();  
        vol = mybox2.volume();  
        System.out.println("Volume of mybox2 is " + vol);  
        System.out.println("Weight of mybox2 is " + mybox2.weight);  
    }  
}
```

The output from this program is shown here:

Volume of mybox1 is 3000.0

Weight of mybox1 is 34.3

Volume of mybox2 is 24.0

Weight of mybox2 is 0.076

Method Overriding

- 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 or using object of 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.
- Method overriding is used for runtime polymorphism.

Method Overriding

- The benefit of overriding is: Ability to define a behavior that's specific to the subclass type. Which means a subclass can implement a superclass method based on its requirement.
- In object oriented terms, overriding means to override the functionality of any existing method.

Rules for Method Overriding

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same as the return type declared in the original overridden method in the superclass.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- Constructors cannot be overridden.

Method Overriding

```
//Java Program to illustrate the use of Java Method Overriding
```

```
//Creating a parent class.
```

```
class Vehicle{
```

```
    //defining a method
```

```
    void run(){System.out.println("Vehicle is running");}
```

```
}
```

```
//Creating a child class
```

```
class Bike2 extends Vehicle{
```

```
    //defining the same method as in the parent class
```

```
    void run(){System.out.println("Bike is running safely");}
```

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

```
    Bike2 obj = new Bike2();//creating object
```

```
    obj.run();//calling method
```

```
}
```

```
}
```

**Bike is running
safely**

Method Overriding

```
class Bank{
int getRateOfInterest(){return 0;}
}
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}
class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
}
```

SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9

super keyword

- The super keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.
- Usage of Java super Keyword :-
- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super can be used to invoke immediate parent class constructor.
- super can be used to differentiate the members of superclass from the members of subclass, if they have same names.

Using super keyword

- *super has two general forms.*
 - To explicitly call the no-argument(default) and parameterized constructor of parent class
 - To access the data members of parent class when both parent and child class have member with same name
 - To access the method of parent class when child class has overridden that method.

super is used to refer immediate parent class instance variable.

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

black
white

super can be used to invoke parent class method

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

eating...
barking...

super is used to invoke parent class constructor

```
class Animal{
    Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
    Dog(){
        super();
        System.out.println("dog is created");
    }
}
class TestSuper3{
    public static void main(String args[]){
        Dog d=new Dog();
    }
}
```

animal is created
dog is created

super example: real use

```
class Person{
    int id;
    String name;
    Person(int id,String name){
        this.id=id;
        this.name=name;
    }
}

class Emp extends Person{
    float salary;
    Emp(int id,String name,float salary){
        super(id,name);//reusing parent constructor
        this.salary=salary;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}

class TestSuper5{
    public static void main(String[] args){
        Emp e1=new Emp(1,"ankit",45000f);
        e1.display();
    }
}
```

1 ankit 45000

Using super to Call Superclass Constructors

- A subclass can call a constructor method defined by its superclass by use of the following form of **super**:

super(parameter-list);

 - *parameter-list specifies any parameters needed by the constructor in the superclass.*
- **super()** must always be the first statement executed inside a subclass' constructor.

Extending Box class using Super

// Here, Box is extended to include weight.

```
class BoxWeight extends Box {  
    double weight; // weight of box  
    // constructor for BoxWeight  
    BoxWeight(double w, double h, double d, double m) {  
        super(w, h, d); // call superclass constructor  
        weight = m;  
    }  
}
```

// Here, Box is extended to include color.

```
class ColorBox extends Box {  
    int color; // color of box  
    ColorBox(double w, double h, double d, int c) {  
        super(w, h, d); // call superclass constructor  
        color = c;  
    }  
}
```

A Second Use for super

- Super always refers to the superclass of the subclass in which it is used.
- This usage has the following general form:

`super.member`

- *member can be either a method or an instance variable.*

Example

```
// Using super to overcome name hiding.
class A {
    int i;
}
class B extends A {    // Create a subclass by extending class A.
    int i; // this i hides the i in A
    B(int a, int b) {
        super.i = a; // i in A
        i = b; // i in B
    }
    void show() {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
    }
}
class UseSuper {
    public static void main(String args[]) {
        B subOb = new B(1, 2);
        subOb.show();
    }
}
```

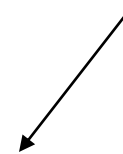
When Constructors are called

- When a class hierarchy is created, in what order constructors are called?
 - In a class hierarchy, constructors are called in **order of derivation, from superclass to subclass.**
 - `super()` must be the first statement executed in a subclass' constructor
 - If `super()` is not used, then the default or parameter less constructor of each superclass will be executed first in subclass constructor.

Subclasses & Constructors

- Default constructor automatically calls constructor of the base class:

default constructor
for Circle class is
called



```
GraphicCircle drawableCircle = new GraphicCircle();
```

Subclasses & Constructors

- Defined constructor can invoke base class constructor with *super*:

```
public GraphicCircle(double x, double y, double r,  
Color outline, Color fill) {  
    super(x, y, r);  
    this.outline = outline;  
    this.fill = fill;  
}
```

Example

```
// Method overriding.
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }
    // display i and j
    void show() {
        System.out.println("i and j:
            " + i + " " + j);
    }
}

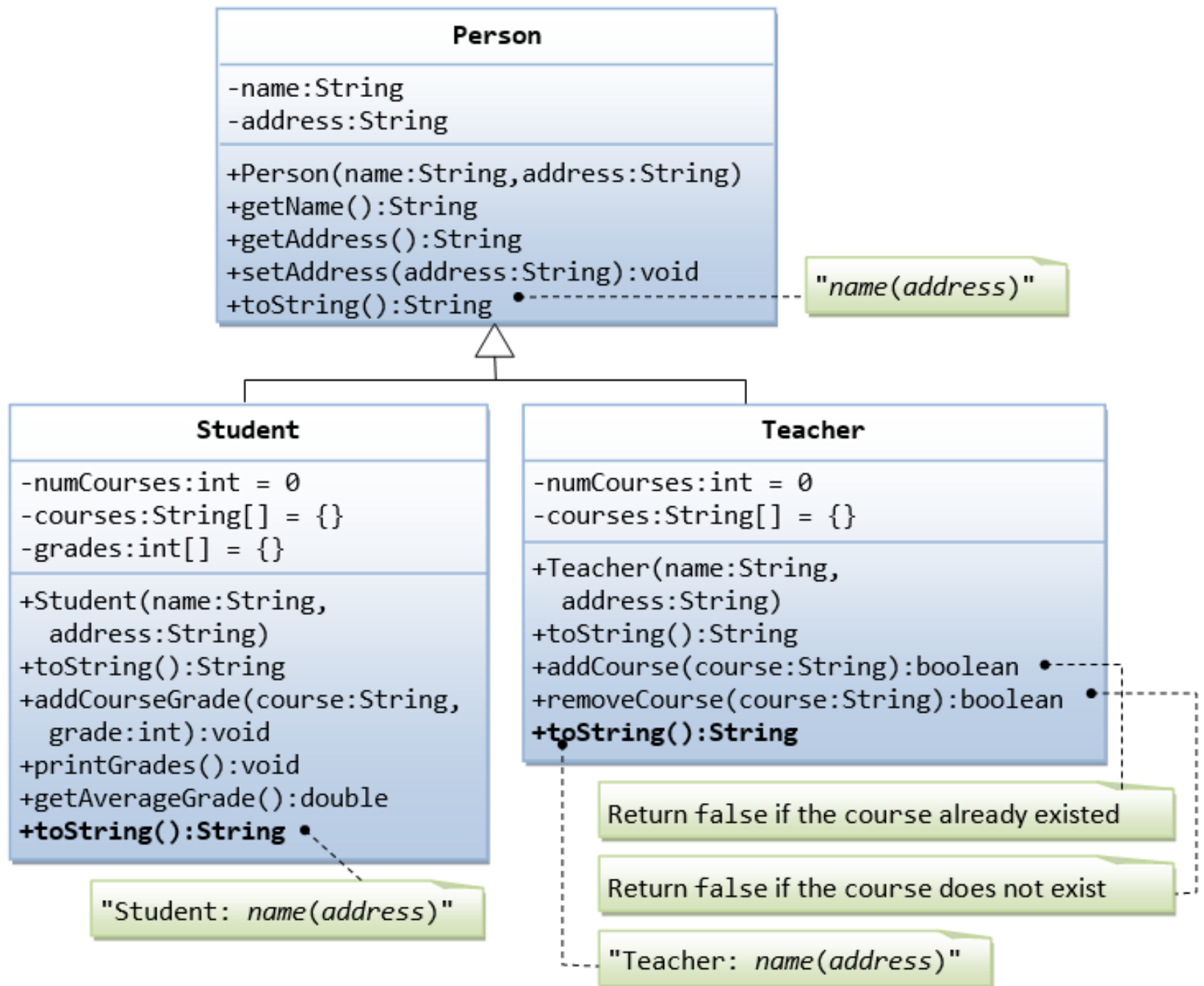
class B extends A {
    int k;
    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
}
```

```
//display k-this overrides show() in A
    void show() {
        System.out.println("k: " + k);
    }
}

class Override {
    public static void main(String args[])
    {
        B subOb = new B(1, 2, 3);
        subOb.show(); //this calls
                       show()in B
    }
}
```

Example

- Suppose that we are required to model students and teachers in our application.
 - Define a superclass called Person to store common properties such as name and address
 - Design subclasses Student and Teacher for their specific properties.
 - For students, maintain the no of courses taken, name of courses and their respective grades; method to add a course with grade, method to print all courses taken and the average grade. Assume that a student takes no more than 30 courses for the entire program.
 - For teachers, maintain the courses names and no of courses taught currently, and method to add or remove a course taught. Assume that a teacher teaches not more than 5 courses concurrently.



Final Keyword

- The final keyword in java is used to restrict the user.
- The final keyword is a non-access modifier used for classes, attributes and methods, which makes them non-changeable (impossible to inherit or override).
- The final keyword is useful when you want a variable to always store the same value, like PI (3.14159...).
- The java final keyword can be used with:
 - **variable**
 - **method**
 - **class**

Final Keyword

- **Final Variable:** If you make any variable as final, you cannot change the value of that final variable (It will be constant).
- A variable that is declared as final and not initialized is called a blank final variable. A blank final variable forces the constructors to initialize it.
- **Final Method:** Methods declared as final cannot be overridden.
- **Final Class:** Java classes declared as final cannot be extended means cannot inherit.
- If you declare any parameter as final, you cannot change the value of it.

Final Classes

- Declaring class with *final* modifier prevents it being extended or subclassed.
- Allows compiler to optimize the invoking of methods of the class

```
final class Circle{  
    .....  
  
}
```

Final Keyword

Final variable:

- If you make any variable as final, you cannot change the value of final variable(It will be constant).
- There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{  
    final int speedlimit=90;//final variable  
    void run(){  
        speedlimit=400;  
    }  
    public static void main(String args[]){  
        Bike9 obj=new Bike9();  
        obj.run();  
    }  
}//end of class
```

**Output: Compile Time
Error**

Final Keyword

Final method:

- If you make any method as final, you cannot override it.

```
class Bike{  
    final void run(){System.out.println("running");}  
}  
  
class Honda extends Bike{  
    void run(){System.out.println("running safely with 100kmph");}  
  
    public static void main(String args[]){  
        Honda honda= new Honda();  
        honda.run();  
    }  
}
```

Output: Compile Time Error

Final Keyword

Final class:

- If you make any class as final, you cannot extend it.

```
final class Bike{
```

```
class Honda1 extends Bike{
```

```
void run(){System.out.println("running safely with 100kmph");}
```

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

```
Honda1 honda= new Honda1();
```

```
honda.run();
```

```
}
```

```
}
```

**Output: Compile Time
Error**

Polymorphism

- The word "*polymorphism*" means "*many forms*". It comes from Greek word "*poly*" (means *many*) and "*morphos*" (means *form*).
- It is a concept in OOP that allows objects of different classes to be treated as if they are objects of the same class, thus providing a mechanism to write a code that can work with objects of different classes without knowing their specific types at compile time.

Polymorphism

- There are two types of polymorphism in Java:
- **compile-time polymorphism (method overloading)**
- **runtime polymorphism (method overriding)**

Polymorphism

- **compile-time polymorphism:-**
- This type of polymorphism in Java is also called static polymorphism or static method dispatch.
- It can be achieved by method overloading.
- In this process, an overloaded method is resolved at compile time rather than resolving at runtime.
- The same method name will be used with different number of parameters and parameters of different type.
- Overloading of methods with different return types is not allowed.

Polymorphism

```
class overloadingDemo
{
void sum(int a,int b)
{
System.out.println("Sum of (a+b) is:: "+(a+b));
}
void sum(int a,int b,int c)
{
System.out.println("Sum of (a+b+c) is:: "+(a+b+c));
}
void sum(double a,double b)
{
System.out.println("Sum of double (a+b) is:: "+(a+b));
}
public static void main(String args[])
{
    overloadingDemo o1 = new overloadingDemo();
        o1.sum(10,10);           // call method1
        o1.sum(10,10,10);       // call method2
        o1.sum(10.5,10.5);      // call method3
    }
}
```

Sum of (a+b) is:: 20

Sum of (a+b+c) is:: 30

Sum of double (a+b) is:: 21.0

Run Time Polymorphism

- Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch*.
- *Dynamic* method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Java implements run-time polymorphism.
- In a Class hierarchy when an overridden method is called through a superclass reference variable.
- A superclass reference variable can refer to a subclass object. It is known as **Upcasting**.
- When an overridden method is called through a superclass reference, the determination of the method to be called is based on the object being referred to by the reference variable.
- This determination is made at run time.

Dynamic Method Dispatch

- Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs (**run time**).
- *it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.*
- if a superclass contains a method that is overridden by a subclasses.
- when different types of objects are referred to through a superclass reference variable, different versions of the method are executed (**Polymorphism**).

Example

// Dynamic Method Dispatch

```
class A {  
    void callme() {  
        System.out.println("Inside A's callme method");  
    }  
}  
  
class B extends A {  
    // override callme()  
    void callme() {  
        System.out.println("Inside B's callme method");  
    }  
}  
  
class C extends A {  
    // override callme()  
    void callme() {  
        System.out.println("Inside C's callme method");  
    }  
}
```

Example

```
class Dispatch {  
    public static void main(String args[]) {  
        A a = new A(); // object of type A  
        B b = new B(); // object of type B  
        C c = new C(); // object of type C  
        A r; // obtain a reference of type A  
        r = a; // r refers to an A object  
        r.callme(); // calls A's version of callme  
        r = b; // r refers to a B object  
        r.callme(); // calls B's version of callme  
        r = c; // r refers to a C object  
        r.callme(); // calls C's version of callme  
    }  
}
```

Output:

Inside A's callme method

Inside B's callme method

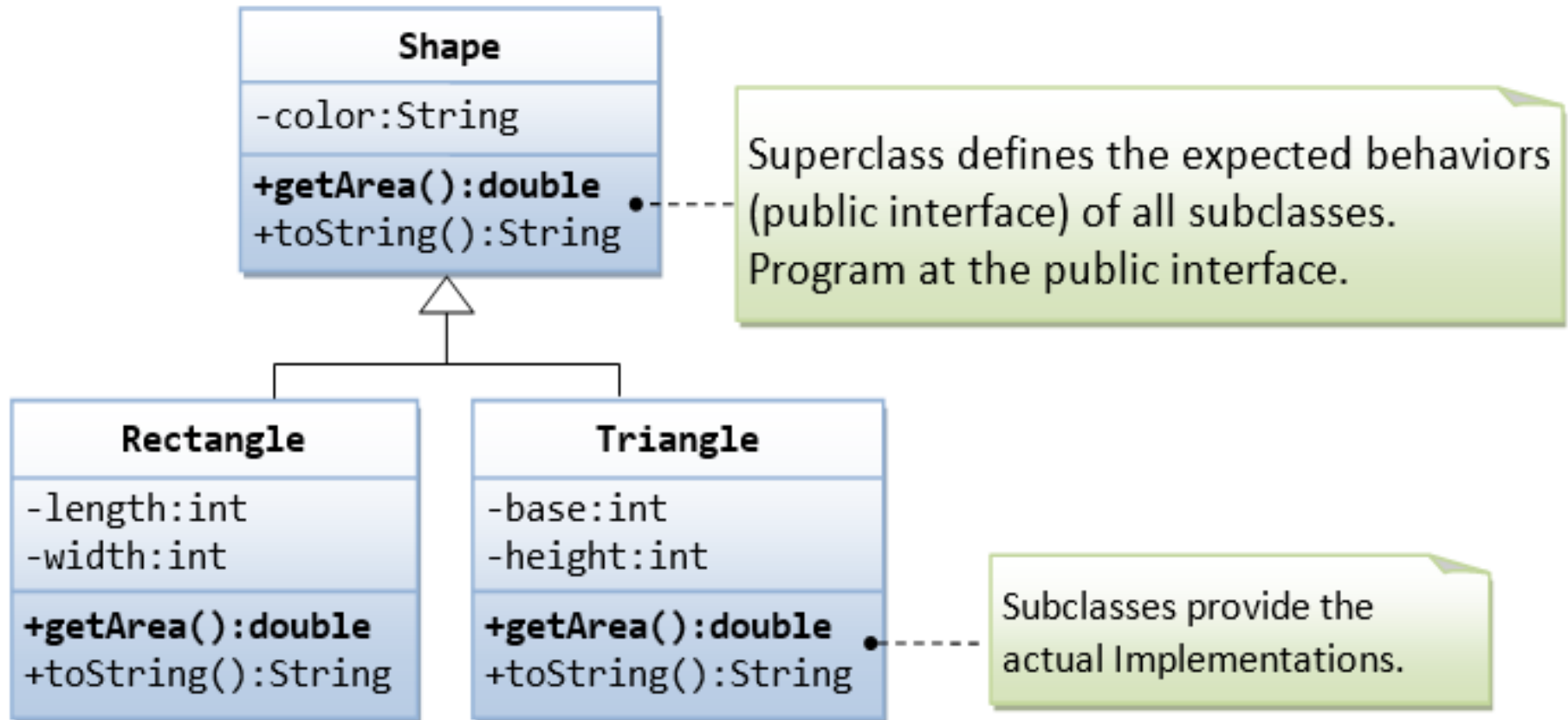
Inside C's callme method

Polymorphism

Overloading	Overriding
Method overloading is used <i>to increase the readability</i> of the program.	Method overriding is used <i>to provide the specific implementation</i> of the method that is already provided by its super class.
Method overloading is performed <i>within class</i> .	Method overriding occurs <i>in two classes</i> that have IS-A (inheritance) relationship.
In case of method overloading, <i>parameter must be different</i> .	In case of method overriding, <i>parameter must be same</i> .
Method overloading is the example of <i>compile time polymorphism</i> .	Method overriding is the example of <i>run time polymorphism</i> .
In java, method overloading can't be performed by changing return type of the method only. <i>Return type can be same or different</i> in method overloading.	<i>Return type must be same or covariant</i> in method overriding.
Private and final methods can be overloaded.	Private and final methods cannot be overridden.

Example

Suppose that our program uses many kinds of shapes, such as triangle, rectangle and so on. Design a superclass called Shape, which defines the public interfaces (or behaviors) of all the shapes. For example, method called `getArea()`, which returns the area of that particular shape



Superclass Shape.java

```
/* * Superclass Shape maintain the common properties of all shapes */
public class Shape { // Private member variable
    private String color;
    // Constructor
    public Shape (String color) {
        this.color = color;
    }

    @Override
    public String toString() {
        return "Shape[color=" + color + "]";
    }

    // All shapes must have a method called getArea().

    public double getArea() { // We have a problem here!
    // We need to return some value to compile the program.
        System.err.println("Shape unknown! Cannot compute area!");
        return 0;
    }
}
```


Subclass Rectangle.java

```
/* * The Rectangle class, subclass of Shape */
public class Rectangle extends Shape {
    // Private member variables private int length; private int width;
    // Constructor
    public Rectangle(String color, int length, int width) {
        super(color);
        this.length = length;
        this.width = width;
    }
    @Override
    public String toString() {
        return "Rectangle[length=" + length + ",width=" + width + ","
            + super.toString() + "]\n";
    }
    // Override the inherited getArea() to provide the proper
    //implementation
    @Override
    public double getArea(){
        return length*width;
    }
}
```

Subclass Triangle.java

```
/* * The Triangle class, subclass of Shape */
public class Triangle extends Shape {
    // Private member variables
    private int base;
    private int height;
    // Constructor
    public Triangle(String color, int base, int height) {
        super(color);
        this.base = base;
        this.height = height;
    }
    @Override
    public String toString() {
        return "Triangle[base=" + base + ",height=" + height
            + "," + super.toString() + "];"
    } // Override the inherited getArea() to provide the proper
    //implementation
    @Override
    public double getArea() {
        return 0.5*base*height;
    }
}
```

TestShape.java

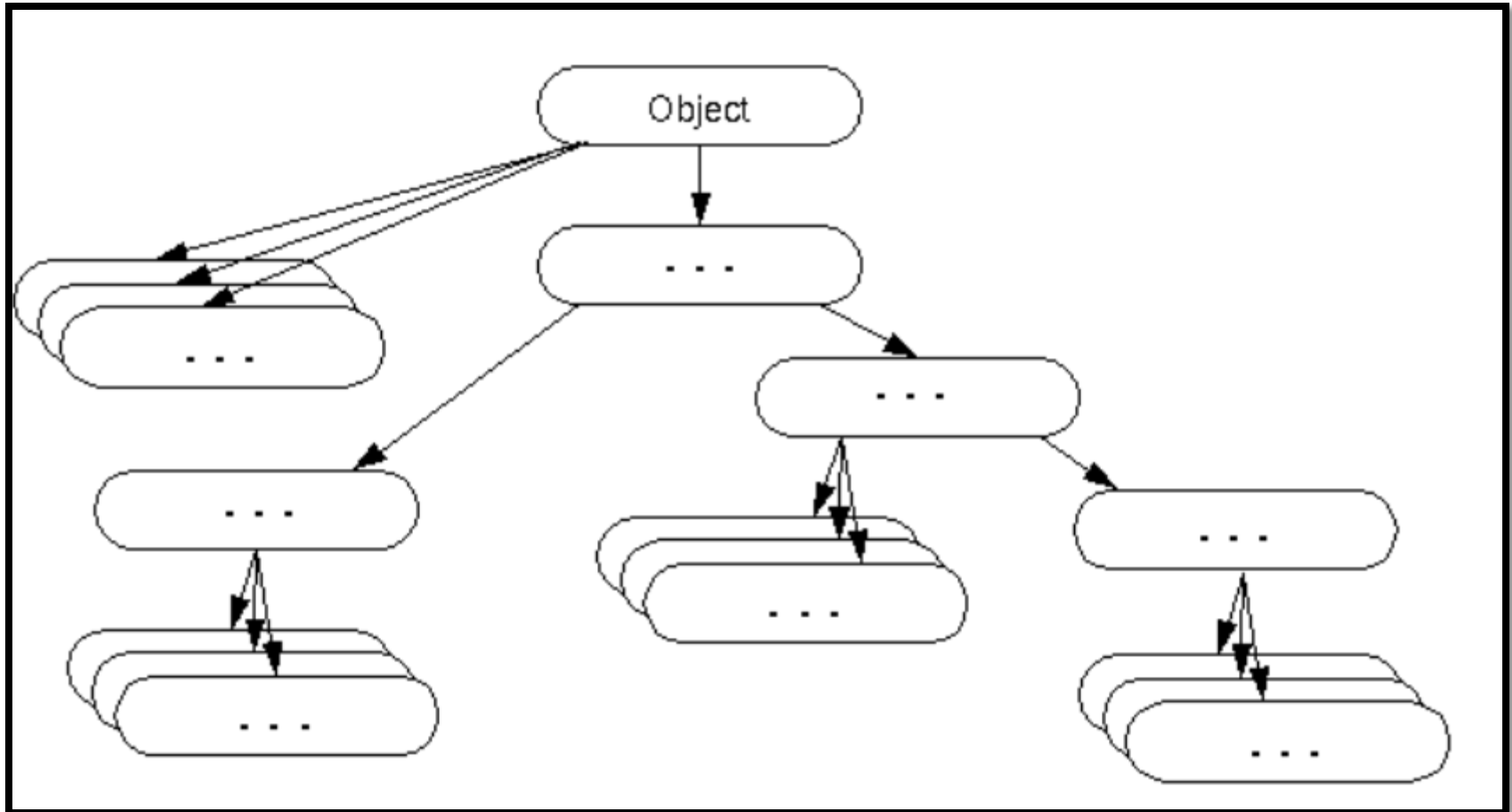
```
/* * A test driver for Shape and its subclasses */

public class TestShape {
    public static void main(String[] args) {
        Shape s1 = new Rectangle("red", 4, 5); //Upcast
        //Run Rectangle's toString ()
        System.out.println(s1);
        // Run Rectangle's getArea()
        System.out.println("Area is " + s1.getArea());
        Shape s2 = new Triangle("blue", 4, 5); // Upcast
        // Run Triangle's toString()
        System.out.println(s2);
        // Run Triangle's getArea()
        System.out.println("Area is " + s2.getArea());
    }
}
```

The Object Class

- The Object class is the parent class (`java.lang.Object`) of all the classes in java by default.
- It means that all the classes in Java are derived classes and their base class is the Object class.
- The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, known as Upcasting.
- The Object class provides some common behaviors to all the objects must have, such as object can be compared, object can be converted to a string, object can be notified etc..

The Object Class



Method	Purpose
Object clone()	Creates a new object that is the same as the object being cloned.
boolean equals(Object object)	Determines whether one object is equal to another.
void finalize()	Called before an unused object is recycled.
Class getClass()	Obtains the class of an object at run time.
int hashCode()	Returns the hash code associated with the invoking object.
void notify()	Resumes execution of a thread waiting on the invoking object.
void notifyAll()	Resumes execution of all threads waiting on the i nvoking object.
String toString()	Returns a string that describes the object.
void wait() void wait(long illiseconds) void wait(long illiseconds, int nanoseconds)	Waits on another thread of execution. getclass, notify, notifyall and wait are final methods

The Object Class

```
class parent
{
    inti = 10;
    Integer i1 = new Integer(i);
    void PrintClassName(Object obj) // Pass object of class as an argument
    {
        System.out.println("The Object's class name is ::" + obj.getClass().getName());
    }
}

class ObjectClassDemo
{
    public static void main(String args[])
    {
        parent a1 = new parent();
        a1.PrintClassName(a1);
        System.out.println("String representation of object i1 is:: "+a1.i1.toString());
    }
}

Output:
The Object's class name is :: parent
String representation of object i1 is:: 10
```

Abstract Class

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- There are two ways to achieve abstraction in java
- Abstract class (0 to 100%)
- Interface (100%)

Abstract Class

- A class which is declared with the abstract keyword is known as an abstract class in Java.
- It can have abstract and non-abstract methods (method with the body).
- It needs to be extended and its method implemented.
- An abstract class cannot be directly **instantiated** with the new operator. (no object of abstract class).
- To declare a class abstract, use the **abstract** keyword in front of the class declaration.
- Abstract class can have normal methods, constructor and instance variable.
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract**.

Abstract Class

Features:-

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Example of abstract class

- **abstract class A{}**

Abstract Class

Abstract Method:-

- A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method:-

- **abstract void printStatus();**//no method body and abstract

Abstract Class

```
abstract class Bike{  
    abstract void run();  
}  
class Honda4 extends Bike{  
    void run(){System.out.println("running safely");}  
    public static void main(String args[]){  
        Bike obj = new Honda4();  
        obj.run();  
        running safely  
    }  
}
```

Abstract Class

```
abstract class Shape{  
abstract void draw();  
}
```

//In real scenario, implementation is provided by others i.e. unknown by end user

```
class Rectangle extends Shape{  
void draw(){System.out.println("drawing rectangle");}  
}
```

```
class Circle1 extends Shape{  
void draw(){System.out.println("drawing circle");}  
}
```

//In real scenario, method is called by programmer or user

```
class TestAbstraction1{  
public static void main(String args[]){
```

```
Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() method  
s.draw();  
}  
}
```

drawing circle

Abstract Class

//Example of an abstract class that has abstract and non-abstract methods

```
abstract class Bike{  
    Bike(){System.out.println("bike is created");}  
    abstract void run();  
    void changeGear(){System.out.println("gear changed");}  
}
```

//Creating a Child class which inherits Abstract class

```
class Honda extends Bike{  
    void run(){System.out.println("running safely..");}  
}
```

//Creating a Test class which calls abstract and non-abstract methods

```
class TestAbstraction2{  
    public static void main(String args[]){  
        Bike obj = new Honda();  
        obj.run();  
        obj.changeGear();  
    }  
}
```

**bike is created
running safely..
gear changed**

Interface

- An interface is a collection of abstract methods and constants that can be implemented by any class.
- An interface is not a class.
- When you create an interface it defines what a class can do without saying anything about how the class will do it.
- Interface contains only static constants and abstract methods only.
- The interface in java is a mechanism to achieve fully abstraction.
- There can be only abstract methods in the java interface not method body.

Interface

- By default (Implicitly), an interface is abstract, interface fields(data members) are public, static and final and methods are public and abstract.
- It is used to achieve fully abstraction and multiple inheritance in Java.
- Similarity between class and interface are given below:
- **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 bytecode of an interface appears in a .class file.**

Class v/s Interface

Class	Interface
In class, you can instantiate variables and create an object.	In an interface, you can't instantiate variables and create an object.
Class can contain concrete(with implementation) methods	The interface cannot contain concrete(with implementation) methods
The access specifiers used with classes are private, protected, and public.	In Interface only one specifier is used-Public.
It contains default as well as parameterize constructors.	It does not contain any constructors.
A class can inherit only one Class and can implement many interfaces.	An interface cannot inherit any class while it can extend many interfaces.

Rules for using Interface

- Consists of only abstract methods and final variables.
- Any number of classes can implement an interface.
- One class can implement any number of interfaces.
- Interface can not implement class.
- Interface can be nested inside another interface.
- To implement an interface a class must define each of the method declared in the interface. Each class can also add new features.
- All variables declared inside interface are implicitly public static final variables(constants).

Defining an Interface

- General form of an interface:

```
access interface name {  
    //Any number of final, static fields  
    //Any number of abstract method declarations  
    type  
}
```

Example:

```
public interface callback{  
    int i=10;  
    void callback (int param);  
}
```

Defining an Interface

- *The interface keyword is used to declare an interface.*
- Access is either **public** or **default**.
- Variables declared inside an interface are implicitly **final** and **static**.
- Variables must be initialized with a constant value.
- All methods and variables are implicitly **public** if the interface, itself, is declared as **public**.

Interface Syntax

```
[public|protected|package]interface  
interfaceName[extends superInterfaceName]{  
    // constants  
    static final ...;  
    // public abstract methods' signature  
    ...  
}
```

Implementing Interfaces

- A class uses the implements keyword to implement an interface.
- A class implements an interface means, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface.
- If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

Implementing Interfaces

- The General Form:

```
access class classname [implements interface-name]  
{  
    implements or defines all the abstract methods  
}
```

- The methods that implement an interface must be declared *public*.
- The type signature of the implementing method must match exactly the type signature specified in the *interface* definition.

Example-1

```
interface call
{
    void callback(int param);
}

class client implements call
{
    public void callback(int p)
    {
        System.out.println("callback called with "+ p);
    }
}

public class testIface
{
    public static void main(String args[])
    {
        call c = new client();
        c.callback(423);
    }
}
```


Example-2

```
interface call{
    void callback(int param);
}

class client implements call{
    public void callback(int p){
        System.out.println("callback
                           is called with "+p);
    }
}

class anotherclient implements call{
    public void callback(int p){
        System.out.println("p squared
                           is "+(p*p));
    }
}
```

```
public class testIface
{
    public static void
        main(String args[]){
        call c = new client();
        c.callback(42);
        c=new anotherclient();
        c.callback(10);
    }
}
```

Output:

```
callback called with 42
Another version of callback
p squared is 100
```

Example-3

//Interface declaration: by first user

```
interface Drawable{
```

```
void draw();
```

```
}
```

//Implementation: by second user

```
class Rectangle implements Drawable{
```

```
public void draw(){System.out.println("drawing rectangle");}
```

```
}
```

```
class Circle implements Drawable{
```

```
public void draw(){System.out.println("drawing circle");}
```

```
}
```

//Using interface: by third user

```
class TestInterface1{
```

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

```
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
```

```
d.draw();
```

```
}}
```

drawing circle

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.
Abstract class can have static methods, main method and constructor.	Interface can't have static methods, main method or constructor.
Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
They are useful in a situation where specialization behavior should be implemented by child class.	Interfaces are useful in a situation where all properties should be implemented.

Extending Interfaces

- One interface can inherit another by using the keyword **extends**.
- The new sub interface will inherit all the member of the super interface.
- Any class that will implement the interface that inherits another interface, it must provide implementations of all methods defined within the interface inheritance chain.

- General Form:

```
interface name2 extends name1{  
    //body of name2  
}
```

✓An interface cannot extends a class.

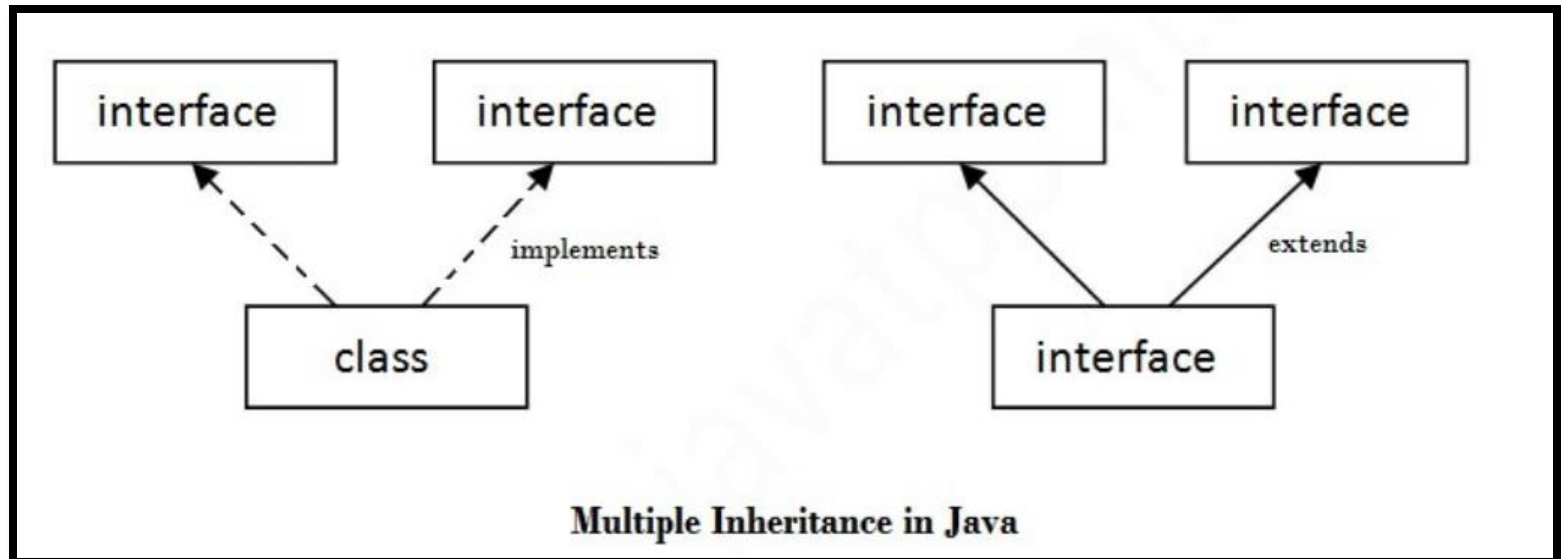
Example:

```
interface ItemConstant{  
    int code =1001;  
    String name ="Pen";  
}
```

```
interface Item extends ItemConstant {  
    void display();  
}
```

Multiple Inheritance Using Interface

- ✓ Java supports multiple inheritance through the use of interface.
- ✓ Care should be taken to avoid some conflicts.
- ✓ If a class implements multiple interfaces or an interface extends multiple interfaces is known as multiple inheritance.



Multiple Inheritance Using Interface

- ✓ A java class can only extend one parent class.
- ✓ Multiple inheritances are not allowed. However, an interface can extend more than one parent interface.
- ✓ The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

Multiple Inheritance Using Interface

```
interface Printable{  
    void print();  
}  
//interface1  
  
interface Showable{  
    void show();  
}  
//interface2  
  
//class implements both interface  
class A7 implements Printable,Showable{  
    public void print(){System.out.println("Hello");}  
    public void show(){System.out.println("Welcome");}  
  
    public static void main(String args[]){  
        A7 obj = new A7();  
        obj.print();  
        obj.show();  
    }  
}
```

Hello
Welcome

Example-4

```
interface test1{  
    int val=10;  
    void display();  
}
```

```
interface test2{  
    int val=20;  
    void display();  
}
```

```
class test3 implements test1, test2{  
    public void display(){  
        System.out.println("In test3");  
        System.out.println(test1.val);  
        System.out.println(test2.val);  
    }  
}
```


Example-5

```
interface test1{
    int val=10;
    void display();
}

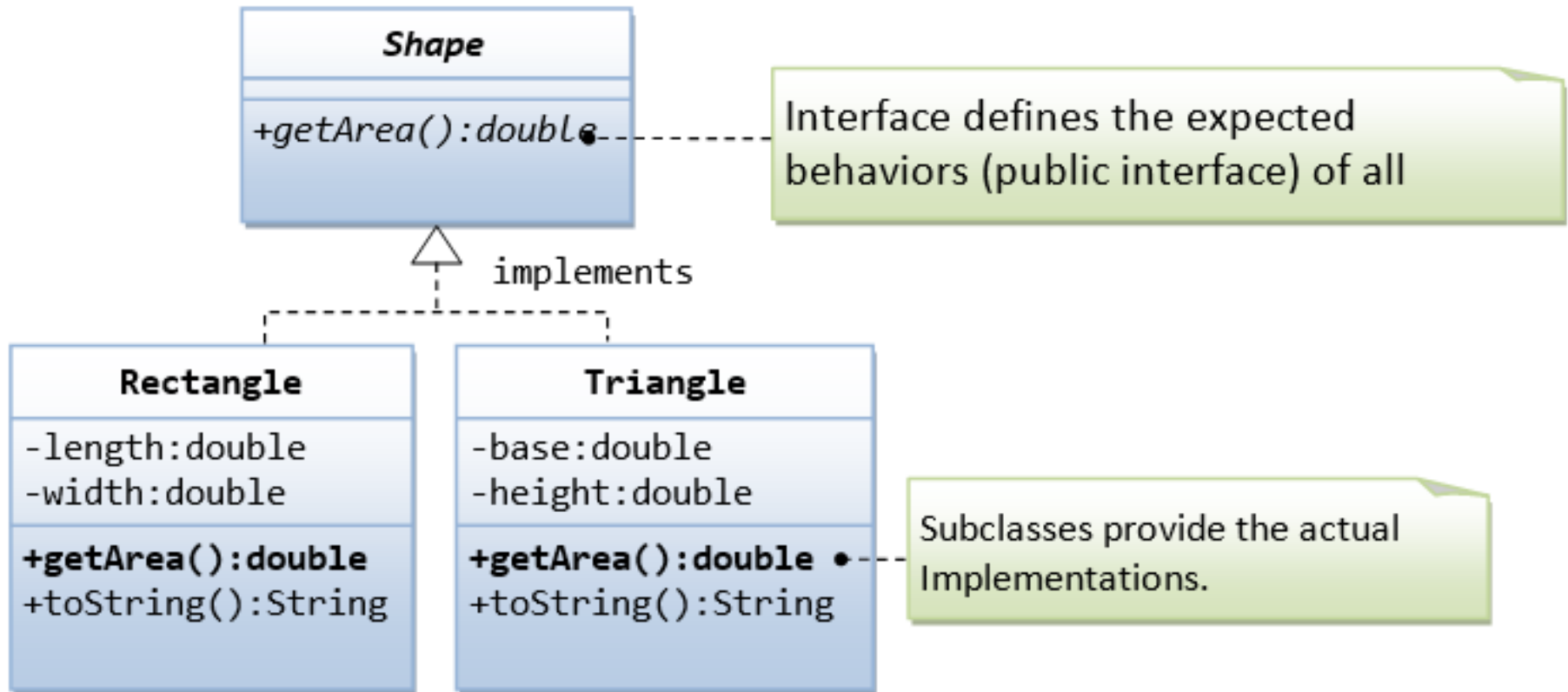
interface test2{
    int val=20;
    void display();
}

interface test3 extends test1,
    test2{
    int val=50;
    void display();
}
```

```
class test4 implements test3{
    int val=57;
    public void display(){
        System.out.println(test1.val);
        System.out.println(test2.val);
        System.out.println(test3.val);
        System.out.println(val);
    }
}

public class Iface_test{
    public static void main(String
        args[]){
        test4 ob = new test4();
        ob.display();
    }
}
```

Shape Interface and its Implementations



Shape Interface and its Implementations

```
/* * The interface Shape specifies the behaviors * of this
implementations subclasses. */
public interface Shape{
    // Use keyword "interface" instead of "class"
    /* List of public abstract methods to be implemented by its
    subclasses */
    //All methods in interface are "public abstract".
    /*"protected", "private" and "package" methods are NOT
    allowed. */
    double getArea();
}
```

Shape Interface and its Implementations

// The subclass Rectangle needs to implement all the abstract methods in Shape

```
public class Rectangle implements Shape{
    // using keyword "implements" instead of "extends"
    // Private member variables
    private int length; private int width;
    // Constructor
    public Rectangle(int length, int width) {
        this.length = length; this.width = width;
    }
    @Override
    public String toString(){
        return "Rectangle[length=" + length + ",width=" + width + "];"
    }
    // Need to implement all the abstract methods defined in the interface
    @Override
    public double getArea(){
        return length * width;
    }
}
```

Shape Interface and its Implementations

```
// The subclass Triangle need to implement all the abstract
methods in Shape
public class Triangle implements Shape{
    private int base; // Private member variables
    private int height;
    public Triangle(int base, int height){    // Constructor
        this.base = base;
        this.height = height;
    }
    @Override
    public String toString(){
        return "Triangle[base="+base+",height=" + height + "]\n";
    } // Need to implement all the abstract methods defined in
the interface
    @Override
    public double getArea(){
        return 0.5 * base * height;
    }
}
```

Shape Interface and its Implementations

```
public class TestShape {  
    public static void main(String[] args){  
        Shape s1 = new Rectangle(1,2); //upcast  
        System.out.println(s1);  
        System.out.println("Area is " + s1.getArea());  
        Shape s2 = new Triangle(3, 4); // upcast  
        System.out.println(s2);  
        System.out.println("Area is " + s2.getArea());  
        // Cannot create instance of an interface  
        //Shape s3 = new Shape("green"); // Compilation Error!!  
    }  
}
```

Default Method

- This feature introduced in Java 8.
- It is used to add new functionality to existing interfaces without breaking compatibility with existing implementation.
- Default methods have an implementation, which means they can provide a default behavior that is used by implementing class unless they provide their own implementation.

Default Method

```
interface vehicle
{
    default void show() // default method
    {
        System.out.println("Default Method Executed");
    }
}
class car implements vehicle
{
    // does not override any method
}
class bike implements vehicle
{
    void show() // default method
    {
        System.out.println("start bike");
    }
}
public class Main
{
    public static void main(String args[])
    {
        vehicle c = new car();
        c.show(); // default method executed
        vehicle d=new car()
        d.show(); // override method
    }
}
```

Default method Executed
Start bike

Default Method

```
// A simple program to Test Interface default methods in java
interface TestInterface
```

```
{
    // abstract method
    public void square(int a);

    // default method
    default void show()
    {
        System.out.println("Default Method Executed");
    }
}
```

```
class TestClass implements TestInterface
```

```
{
    // implementation of square abstract method
    public void square(int a)
    {
        System.out.println(a*a);
    }
    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.square(4);
        // default method executed
        d.show();
    }
}
```

16

Default Method Executed

Lambda Expression

- Lambda expression is a new and important feature of Java which was included in Java SE 8.
- It provides a clear and concise way to represent one method interface using an expression.
- It is very useful in collection library.
- It helps to iterate, filter and extract data from collection.
- The Lambda expression is used to provide the implementation of an interface which has functional interface.
- It saves a lot of code.
- In case of lambda expression, we don't need to define the method again for providing the implementation.
- Java lambda expression is treated as a function, so compiler does not create .class file.

Lambda Expression

- Lambda expression provides implementation of functional interface.
- An interface which has only one abstract method is called functional interface.
- Java provides an annotation `@FunctionalInterface`, which is used to declare an interface as functional interface.

Uses:-

- To provide the implementation of Functional interface.
- Less coding.

Lambda Expression

Syntax:-

(argument-list) -> {body}

- Java lambda expression is consisted of three components.
- 1) Argument-list: It can be empty or non-empty as well.
- 2) Arrow-token: It is used to link arguments-list and body of expression.
- 3) Body: It contains expressions and statements for lambda expression.

Lambda Expression

No Parameter Syntax

```
() -> {  
//Body of no parameter lambda  
}
```

One Parameter Syntax:-

```
(p1) -> {  
//Body of single parameter lambda  
}
```

Two Parameter Syntax

```
(p1,p2) -> {  
//Body of multiple parameter lambda  
}
```

Without Lambda Expression

```
interface Drawable{  
    public void draw();  
}  
  
public class LambdaExpressionExample {  
    public static void main(String[] args) {  
        int width=10;  
  
        Drawing 10  
  
        //without lambda, Drawable implementation using anonymous class  
        Drawable d=new Drawable(){  
            public void draw(){System.out.println("Drawing "+width);}  
        };  
        d.draw();  
    }  
}
```

Lambda Expression

```
@FunctionalInterface //It is optional
interface Drawable{
    public void draw();
}

public class LambdaExpressionExample2 {
    public static void main(String[] args) {
        int width=10;

        Drawing 10

        //with lambda
        Drawable d2=()->{
            System.out.println("Drawing "+width);
        };
        d2.draw();
    }
}
```

Lambda Expression

```
interface Addable{
    int add(int a,int b);
}

public class LambdaExpressionExample5{
    public static void main(String[] args) {
        30
        300

        // Multiple parameters in lambda expression
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));

        // Multiple parameters with data type in lambda expression
        Addable ad2=(int a,int b)->(a+b);
        System.out.println(ad2.add(100,200));
    }
}
```


Packages

■ What is a Package:

- A package, like a library, is a collection of classes, and other related entities such as interfaces, errors, exceptions, annotations, and enums.

■ Packages are used for:

- **Organizing** classes and related entities.
- **Managing namespaces** - Each package is a namespace.
- **Resolving naming conflicts.**
 - For example, `com.zzz.Circle` and `com.yyy.Circle` are treated as two distinct classes. Although they share the same classname `Circle`, they belong to two different packages: `com.zzz` and `com.yyy`. These two classes can co-exist and can even be used in the same program via the fully-qualified names.
- **Access control:**
 - Besides `public` and `private`, you can grant access of a class/variable/method to classes within the same package only.

Packages

- **Distributing Java classes:**

- All entities in a package can be combined and compressed into a single file, known as JAR (Java Archive) file, for distribution.

Benefits:

- The classes contained in the packages of other programs can be reused.
- In packages, classes can be unique compared with classes in other packages.
- Packages provides a way to hide classes.

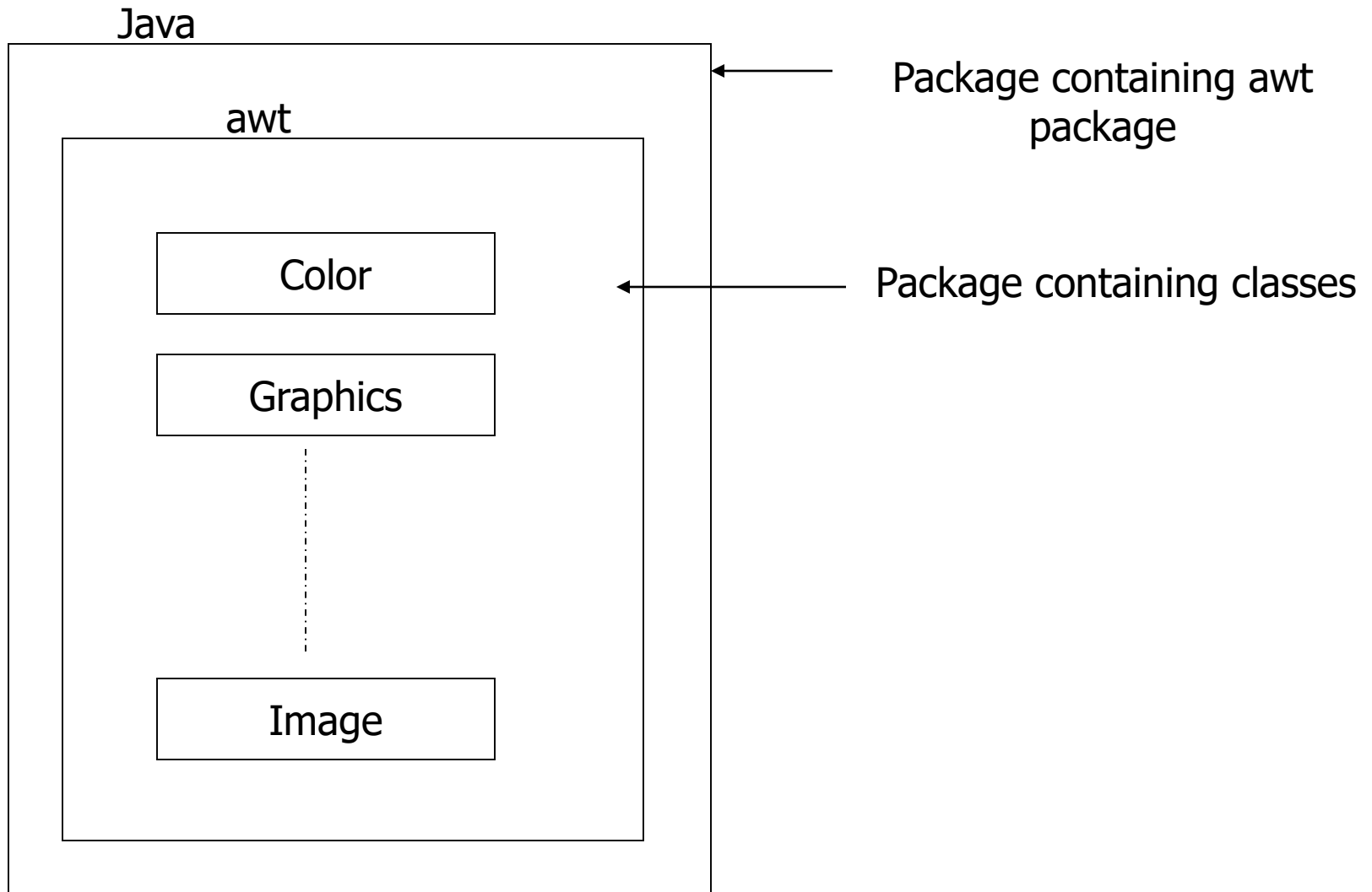
Packages

- Two types of packages:
 - Java API packages (built in)
 - User defined packages

Java API Packages:

- A large number of classes grouped into different packages based on functionality.
- Examples:
 1. java.lang
 2. java.util
 3. java.io
 4. java.awt
 5. java.net
 6. java. applet etc.

Package



Defining Package

- To create a package include a **package** command as the first statement in a Java source file.
- If package statement is omitted, the class names are put into the default package, which has no name.
- General form of the package statement:
 - *package pkg;*
 - *For example :* `package MyPackage;`
- **.class** files for classes declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.
- case is significant, the directory name must match the package name exactly.

Defining Package

- Hierarchy of packages can be created by separating each package name using period.
- The general form of a multileveled package:
 - *package pkg1[.pkg2[.pkg3]];*
- A package hierarchy must be reflected in the file system of your Java development system.
- For example, a package declared as
 - *package java.awt.image;*
 - needs to be stored in **java/awt/image**

Default Package

- ✓ If a source file does not begin with the *package* statement, the classes contained in the source file reside in the *default package*
- ✓ The java compiler automatically looks in the default package to find classes.

Creating Packages

- ✓ Consider the following declaration:

```
package firstPackage.secondPackage;
```

This package is stored in subdirectory named **firstPackage.secondPackage**.

- ✓ A java package can contain more than one class definitions that can be declared as public.
- ✓ Only one of the classes may be declared **public** and that class name with **.java** extension is the source file name.

Steps for Creating Package

1. Declare the package at the beginning of a file using the form

package packagename;

2. Define the class that is to be put in the package and declare it public.
3. Create a subdirectory under the directory where the main source files are stored.
4. Store the listing as **classname.java** in the subdirectory created.
5. Compile the file. This creates **.class** file in the subdirectory.

Example-1

```
// A simple package
package MyPack;
class Balance {
    String name;
    double bal;
    Balance(String n, double b) {
        name = n;
        bal = b;
    }
    void show() {
        if(bal<0)
            System.out.print("->
");
        System.out.println(name + ":
$" + bal);
    }
}
```

```
class AccountBalance {
    public static void main(String args[])
    {
        Balance current[] = new
            Balance[3];
        current[0] = new Balance("K. J.
            Fielding", 123.23);
        current[1] = new Balance("Will
            Tell", 157.02);
        current[2] = new Balance("Tom
            Jackson", -12.33);
        for(int i=0; i<3; i++)
            current[i].show();
    }
}
```

Running Example-1

- Save program with name **AccountBalance.java**, and put it in a directory called **MyPack**.
- compile the file and make sure that the resulting **.class** file is also in the **MyPack** directory.
- execute the **AccountBalance** class as
 - `java MyPack.AccountBalance`
- execute this command from directory above **MyPack** directory.

Accessing Classes in a Package using import statement

- There are two ways to reference a class in your source codes:
 1. Use the *fully-qualified name* in the of from
 - ***packagename.classname*** (such as java.util.Scanner).
 - For example

```
public class ScannerNoImport {  
    public static void main(String[] args) {  
        // Use fully-qualified name in "ALL" the references  
        java.util.Scanner in = new java.util.Scanner(System.in);  
        System.out.print("Enter a integer: ");  
        int number = in.nextInt();  
        System.out.println("You have entered: " + number);  
    }  
}
```

using import statement

2. Add an "import fully-qualified-name" statement at the beginning of the source file.
 - You can then use the classname alone (without the package name) in your source codes.
 - For example

```
import java.util.Scanner;
public class ScannerWithImport{
    public static void main(String[] args) {
        // Package name can be omitted for an imported class
        /* Java compiler searches the import statements for the
           fully-qualified name */
        Scanner in = new Scanner(System.in); // classname only
        System.out.print("Enter a integer: ");
        int number = in.nextInt();
        System.out.println("You have entered: " + number);
    }
}
```

Accessing Classes in a Package

1. Fully Qualified class name:

Example: java.awt.Color

2. **import** packagename.classname; // importing a single class

Example: import java.awt.Color;

or

import packagename.*; // importing all classes in packake using wildcard *

Example: import java.awt.*;

- ✓ Import statement must appear at the top of the file, before any class declaration.

Accessing Classes in a Package

Fully Qualified class name:

- If you use fully qualified name then only declared class of this package will be accessible. So, no need to import the package.
- But, you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have same class name.
- e.g. java.util and java.sql packages contain Date class.

Using `packagename.classname` :

- If you use `packagename.classname` then only declared class of this package will be accessible.
- Syntax: `import packagename.classname;`

Using `packagename.*` :

- If you use `packagename.*` then all the classes and interfaces of this package will be accessible but not subpackages.
- Syntax: `import packagename.*;`

Accessing Classes in a Package

- ✓ The import statement provides us a convenient way for referencing classes without using the fully-qualified name.
- ✓ "Import" does not load the class
- ✓ It only resolves a classname to its fully-qualified name, or brings the classname into the namespace.
- ✓ “import” is strictly a compiled-time activity.
- ✓ The import statement(s) must be placed after the package statement but before the class declaration.
- ✓ The Java core language package java.lang is implicitly imported to every Java program.
- ✓ no explicit import statements are needed for classes inside the java.lang package, such as System, String, Math, Integer and Object.
- ✓ no need for import statements for classes within the same package.

Example2-Package

```
package p1;
public class ClassA
{
    public void displayA( ){
        System.out.println("Class
A");
    }
}
```

Source file – ClassA.java

Subdirectory-p1

ClassA.Java and ClassA.class->p1

```
import p1.*;

class testclass
{
    public static void main(String
str[])
    {
        ClassA obA=new ClassA();
        obA.displayA();
    }
}
```

Source file-testclass.java

testclass.java and testclass.class->in
a directory of which *p1* is
subdirectory.

Example2-Package

```
package p2;
public class ClassB
{
    protected int m =10;
    public void displayB()
    {
        System.out.println("Class
        B");
        System.out.println("m=
        "+m);
    }
}
```

```
import p1.*;
import p2.*;

class PackageTest2
{
    public static void
    main(String str[])
    {
        ClassA obA=new ClassA();
        Classb obB=new ClassB();
        obA.displayA();
        obB.displayB();
    }
}
```

Example 3- Package

```
import p2.ClassB;
class ClassC extends ClassB
{
    int n=20;
    void displayC()
    {
        System.out.println("Class
C");
        System.out.println("m= "+m);
        System.out.println("n= "+n);
    }
}
```

```
class PackageTest3
{
    public static void main(String
args[])
    {
        ClassC obC = new ClassC();
        obC.displayB();
        obC.displayC();
    }
}
```

Example 4- Package

```
classA.java
Package pack;
Public class ClassA
{
    int n=20;
    void displayA()
    {
        System.out.println("Class
A");
    }
}
```

```
ClassB.java
Package mypack;
class classB
{
    public static void main(String
args[])
    {
        pack.classA obA = new
pack.ClassA();//fully qualified
        obA.displayA();
    }
}
```

Package

```
package p1;  
public class Teacher  
{.....}  
public class Student  
{.....}
```

```
package p2;  
public class Courses  
{.....}  
public class Student  
{.....}
```

```
import p1.*;  
import p2.*;  
Student student1; //Error
```

Correct Code:

```
import p1.*;  
import p2.*;
```

```
p1.Student student1;  
p2.Student student2;
```

Finding Packages

✓ Two ways:

1. By default, java runtime system uses current directory as starting point and search all the subdirectories for the package.
2. Specify a directory path using **CLASSPATH** environmental variable.

CLASSPATH Environment Variable

- ✓ The compiler and runtime interpreter know how to find standard packages such as *java.lang* and *java.util*
- ✓ The CLASSPATH environment variable is used to direct the compiler and interpreter to **where user defined packages can be found.**
- ✓ The CLASSPATH environment variable is an ordered list of directories and files

CLASSPATH Environment Variable

- ✓ To set the CLASSPATH variable we use the following command:

```
set CLASSPATH=c:\
```

- ✓ Java compiler and interpreter searches the user defined packages from the above directory.
- ✓ To clear the previous setting we use:

```
set CLASSPATH=
```


Example1-Package[Using CLASSPATH]

```
package p1;  
public class ClassA  
{  
    public void displayA()  
    {  
        System.out.println("Class A");  
    }  
}
```

```
import p1.ClassA;
```

```
Class PackageTest1  
{  
    public static void main(String str[])  
    {  
        ClassA obA=new ClassA();  
        obA.displayA();  
    }  
}
```

Source file – c:\p1\ClassA.java
Compile-javac c:\p1\ClassA.java
Class file in –c:\p1\ClassA.class

Source file-
c:\java\jdk1.6.0_06\bin\PackageTest1.java
Compile-javac PackageTest1.java
Copy –PackageTest1.class -> c:\
Execute-java PackageTest1

Example2-Package[Using CLASSPATH]

```
package p2;
public class ClassB
{
    protected int m =10;
    public void displayB()
    {
        System.out.println("Class B");
        System.out.println("m= "+m);
    }
}
```

Source file – c:\p2\ClassB.java

Compile-c:\p2\ClassB.java

Class file in –c:\p2\ClassB.class

```
import p1.*;
import p2.*;
class PackageTest2
{
    public static void main(String str[])
    {
        ClassA obA=new ClassA();
        Classb obB=new ClassB();
        obA.displayA();
        obB.displayB();} }
```

Source file-

c:\java\jdk1.6.0_06\bin\PackageTest2.java

Compile-javac PackageTest2.java

Copy –PackageTest2.class -> c:\

Execute-java PackageTest2

Example 3- Package[Using CLASSPATH]

```
import p2.ClassB;
class ClassC extends ClassB
{
    int n=20;
    void displayC()
    {
        System.out.println("Class C");
        System.out.println("m= "+m);
        System.out.println("n= "+n);
    }
}
```

Source file – c:\ClassC.java

Compile-c:\ClassC.java

Class file in –c:\ClassC.class

```
class PackageTest3
{
    public static void main(String args[])
    {
        ClassC obC = new ClassC();
        obC.displayB();
        obC.displayC();
    }
}
```

Source file-

c:\java\jdk1.6.0_06\bin\PackageTest3.java

Compile-javac PackageTest3.java

Copy –PackageTest3.class -> c:\

Execute-java PackageTest3

Adding a Class to a Package

- ✓ Every java source file can contain only class declared as **public**.
- ✓ The name of the source file should be same as the name of the public class with **.java** extension.

```
package p1;  
public ClassA{ .....}
```

Source file : ClassA.java

Subdirectory: p1

```
package p1;  
public ClassB{.....}
```

Source file: ClassB.java

Subdirectory:p1

Adding a Class to a Package

1. Decide the name of the package.
2. Create the subdirectory with this name under the directory where the main source file is located.
3. Create classes to be placed in the package in separate source files and declare the package statement.

package packagename;

4. Compile each source file. When completed the package will contain .class files of the source files.

public/package/private scope

- ✓ Scope is concerned with the visibility of program elements such as classes and members
- ✓ Class members (methods or instance fields) can be defined with public, package (default), private or protected scope
- ✓ A class has two levels of visibility:
 - **public** scope means it is visible outside its containing package
 - default scope means it is visible only inside the package. (package scope/ friendly scope)

public/package/private scope

- ✓ A class member with **public** scope means it is visible anywhere its class is visible
- ✓ A class member with **private** scope means it is visible only within its encapsulating class
- ✓ A class/class member with **package** scope means it is visible only inside its containing package
- ✓ A class member with **protected** scope means it is visible every where except the non-subclasses in other package.

Example 1

```
package my_package;

class A      // package scope
{
    // A's public & private members
}

public class B      // public scope
{
    // B's public and private members
}
```


Example-2

```
package my_package;

    class D
    {
        // D's public & private members


        // Class D 'knows' about classes A and B

        private B b;    // OK – class B has public scope
        private A a;    // OK – class A has package scope

    }
```

Example-3

```
package another_package;  
import my_package.*;  
  
class C  
{  
    // C's public & private members  
  
    // class C 'knows' about class B  
  
    private B b;    // OK – class B has public scope  
  
}
```



Example 4

```
package my_package;
    class A
    {
        int get() { return data; }    // package scope
        public A(int d) { data=d;}    // public scope
        private int data;              // private scope
    }

    class B
    {
        void f()
        {
            A a=new A(d);    // OK A has package scope
            int d=a.get();    // OK – get() has package scope
            int d1=a.data;    // Error! – data is private
        }
    }
```

Levels of Access Control

	public	protected	friendly (default)	private
same class	Yes	Yes	Yes	Yes
Subclass in the same package	Yes	Yes	Yes	No
Other class in the same package	Yes	Yes	Yes	No
Subclass in other packages	Yes	Yes	No	No
Non- subclass in other package	Yes	No	No	No

Class Member Access Protection and Package

- The following table shows the access matrix for Java.

Position and Access

Position	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Summary

- Inheritance promotes reusability by supporting the creation of new classes from existing classes.
- Various forms of inheritance can be realised in Java.
- Child class constructor can be directed to invoke selected constructor from parent using super keyword.
- Variables and Methods from parent classes can be overridden by redefining them in derived classes.
- New Keywords: extends, super, final