

# Inheritance

PSG COLLEGE OF TECHNOLOGY, COIMBATORE

DEPARTMENT OF APPLIED MATHEMATICS AND COMPUTATIONAL SCIENCES

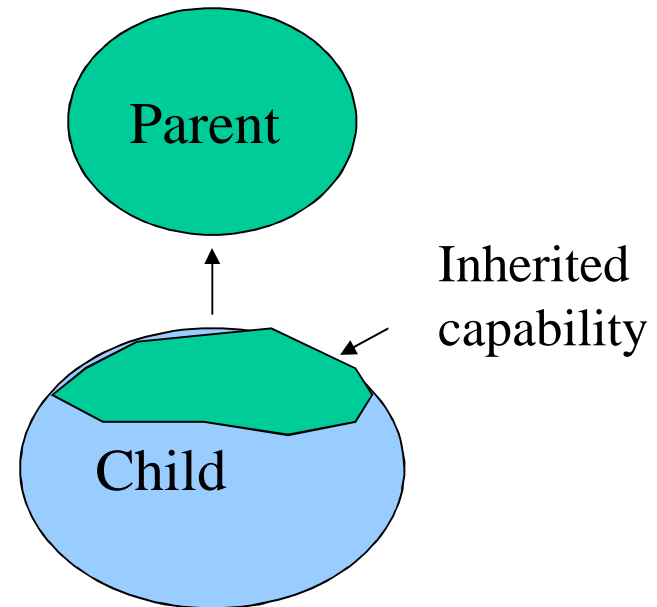
20XW57 – JAVA PROGRAMMING

# Inheritance: Introduction

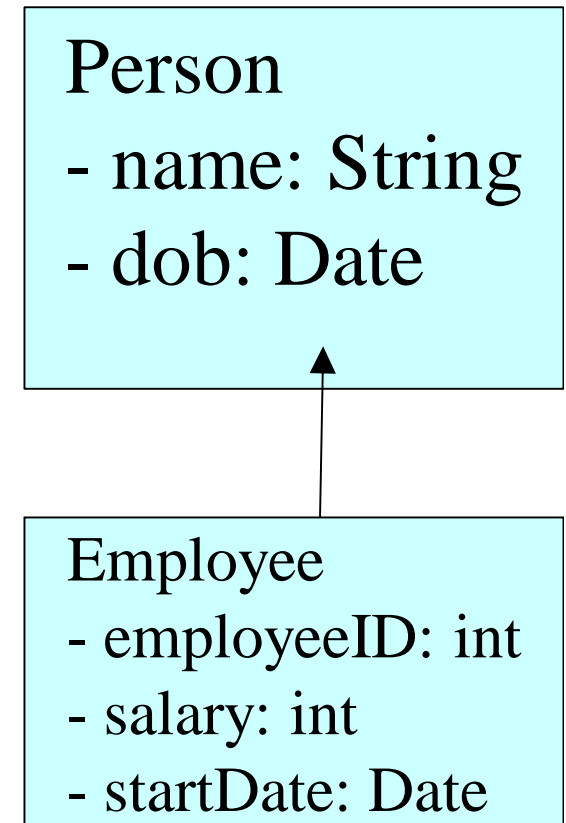
- „ Reusability--building **new components** by utilising **existing components**- is yet another important aspect of OO paradigm.
- „ It is always **good “productive”** if we are able to reuse something that already exists rather than creating the same all over again.
- „ This is achieve by **creating new classes**, **reusing the properties of existing classes**.

# Inheritance: Introduction

- " 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, or “child” class.



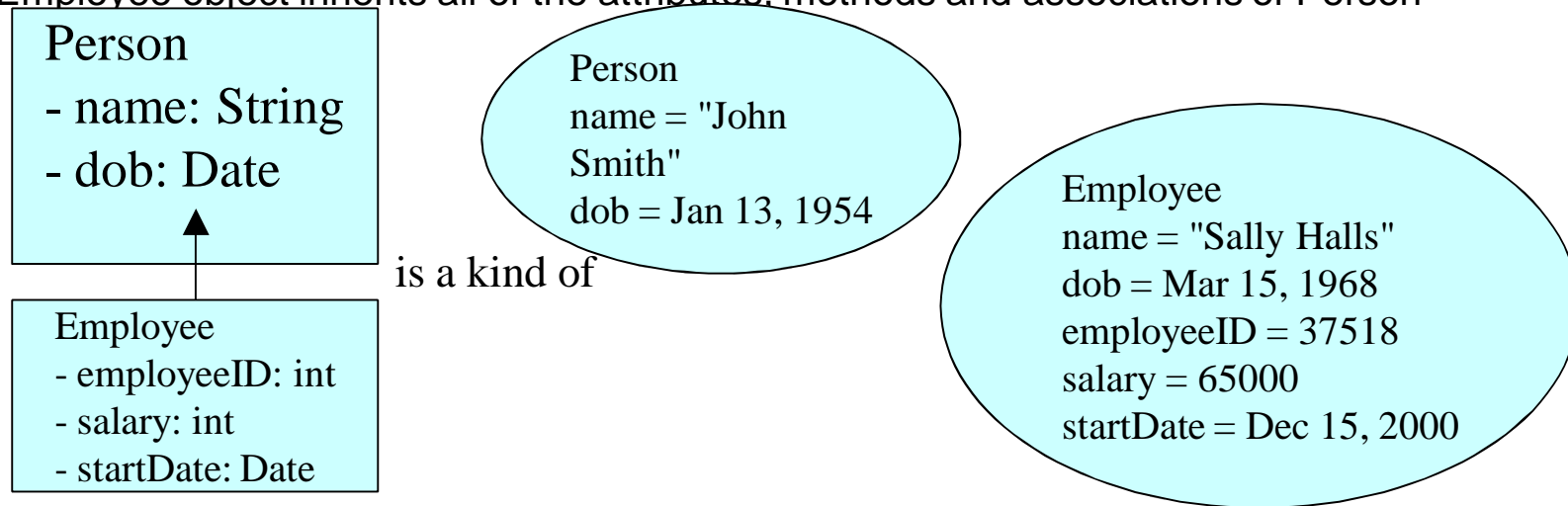
- A class can be defined as a "subclass" of another class.
  - The subclass inherits all data attributes of its superclass
  - The subclass inherits all methods of its superclass
  - The subclass inherits all associations of its superclass
- 
- The subclass can:
    - Add new functionality
    - Use inherited functionality
    - Override inherited functionality



# Memory allocation

---

- When an object is created using new, the system must allocate enough memory to hold all its instance variables.
  - This includes any inherited instance variables
- In this example, we can say that an Employee "is a kind of" Person.
  - An Employee object inherits all of the attributes, methods and associations of Person

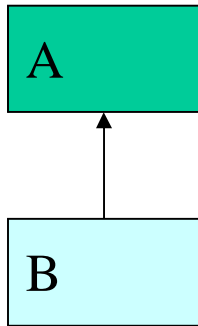


# Inheritance: Introduction

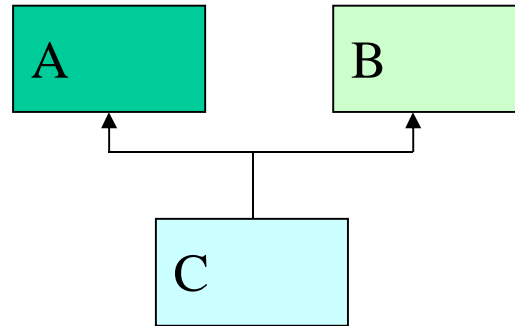
- " The inheritance allows subclasses to inherit all properties (variables and methods) of their parent classes. The different forms of inheritance are:

1. **Single inheritance (only one super class)**
2. **Multiple inheritance (several super classes)**-Not supported by Java-Java enforces a class to have one super class
3. **Hierarchical inheritance (one super class, many sub classes)**
4. **Multi-Level inheritance (derived from a derived class)**
5. **Hybrid inheritance (more than two types)**
6. **Multi-path inheritance (inheritance of some properties from two sources).**

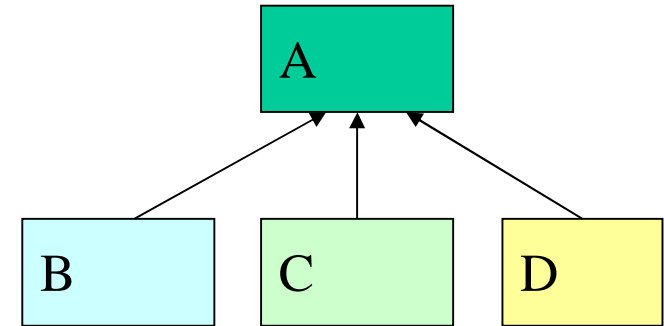
# Forms of Inheritance



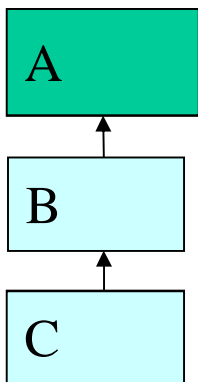
**(a) Single Inheritance**



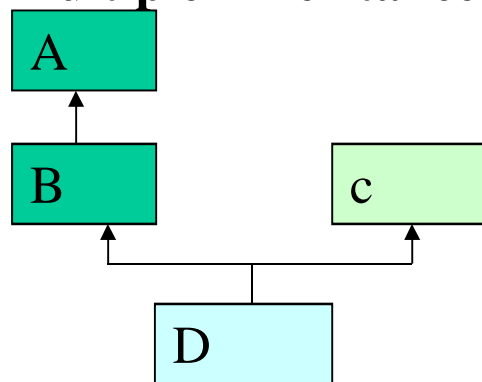
**(b) Multiple Inheritance**



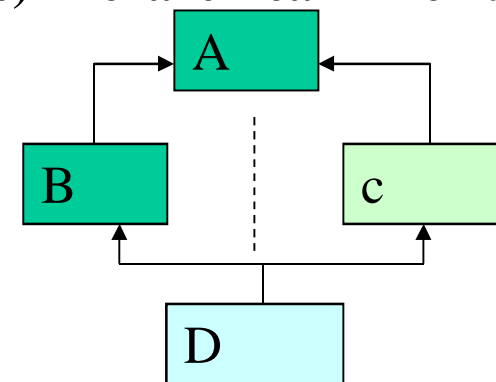
**(c) Hierarchical Inheritance**



**(a) Multi-Level Inheritance**



**(b) Hybrid Inheritance**



**(b) Multipath Inheritance**

# Defining a Sub class

- " A subclass/child class is defined as follows:

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

1. the objects of your class will now receive all of the state (fields) and behavior (methods) of the parent class
2. **constructors and static methods/fields are not inherited**
3. Only public inheritance



## Inheritance in Java

---

- If inheritance is not defined, the class extends a class called Object

```
public class Person
{
    private String name;
    private Date dob;
    [...]
```

```
public class Employee extends Person
{
    private int employeeID;
    private int salary;
    private Date startDate;
    [...]
```

```
Employee anEmployee = new Employee();
```

# Inheritance in Java

1. **Java is a pure object oriented language**
2. **all code is part of some class**
3. **all classes, except one, must inherit from exactly one other class**
4. **The Object class is the *cosmic super class***
  1. The Object class does not inherit from any other class
  2. The Object class is defined in the `java.lang` package
  3. The Object class has several important methods:  
`toString, equals, hashCode, clone, getClass`
5. **implications:**
  1. all classes are descendants of Object
  2. all classes and thus all objects have a `toString, equals, hashCode, clone, and getClass` method
    1. `toString, equals, hashCode, clone` normally overridden

## Visibility of members

Data Fields and Methods	Modifiers			
	public	protected	default	private
Accessible from same class?	yes	yes	yes	yes
Accessible to classes ( <b>nonsubclass</b> ) from the <b>same package</b> ?	yes	yes	yes	no
Accessible to <b>subclass</b> from the <b>same package</b> ?	yes	yes	yes	no
Accessible to classes ( <b>nonsubclass</b> ) from <b>different package</b> ?	yes	no	no	no
Accessible to <b>subclasses</b> from <b>different package</b> ?	yes	no	no	no
Inherited by subclass in the same package?	yes	yes	yes	no

## Class Relationships in Java

```
graph TD; A[Class Relationships in Java] --> B[DEPENDENCE]; A --> C[ASSOCIATION]; A --> D[INHERITANCE]; B --> E[Uses-A Relationship]; C --> F[Has-A Relationship]; F --> G[Aggregation]; F --> H[Composition]; D --> I[Is-A Relationship];
```

**DEPENDENCE**

**Uses-A  
Relationship**

**ASSOCIATION**

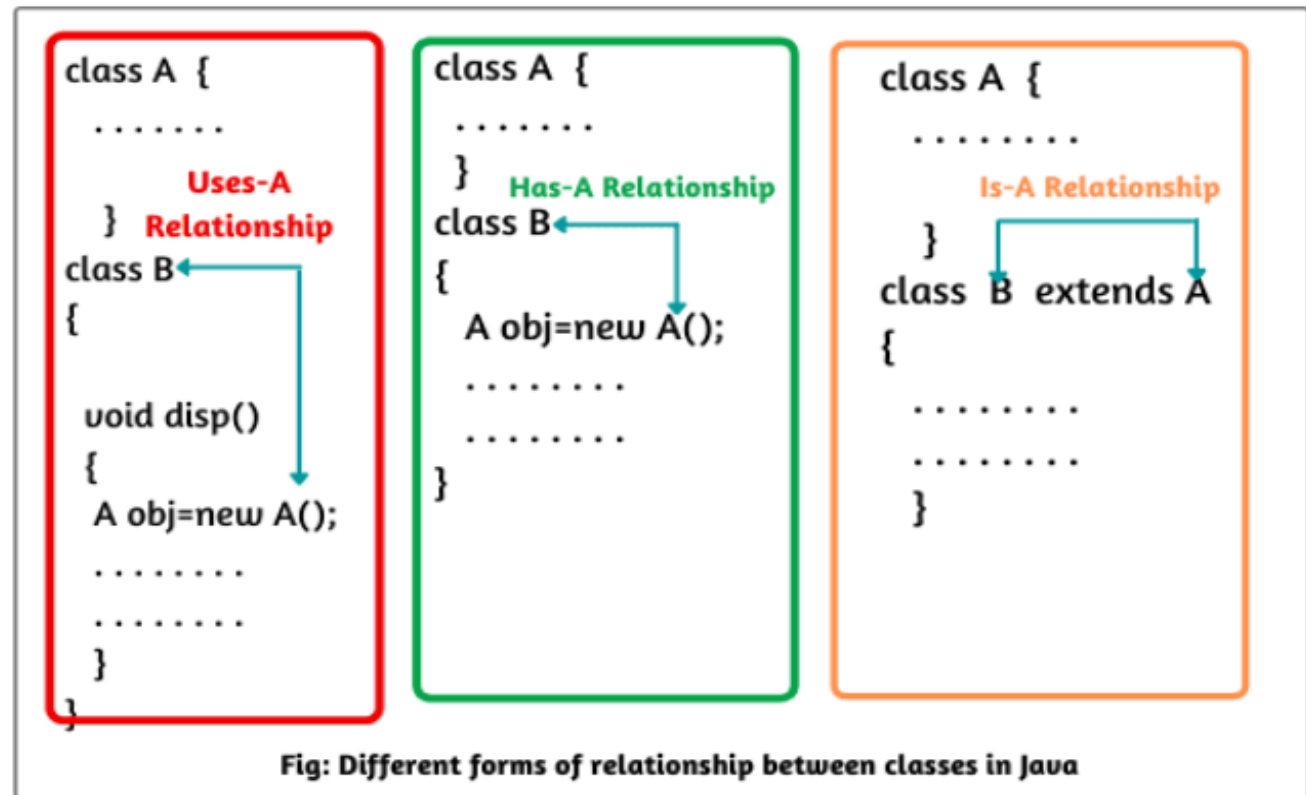
**Has-A  
Relationship**

**Aggregation**

**Composition**

**INHERITANCE**

**Is-A  
Relationship**



- " inheritance defines a relationship between classes, there are instances where relationships between objects exist.
- " These associations in java help the objects of one class to communicate with the objects of the other class.


# Association

- „ Association is a commonly used term in Object-Oriented Programming for both has-a and part-of relationships, but it is not restricted to these.
- „ When we say that two objects are in an association relationship, we are making a generic statement, which means that we are **not concerned with the objects' lifetime dependency.**

# Has a relationship

```
" public class Address
"
" { // Code goes here. }
"
" public class Person
" { // Person has-a Address.
"
" Address addr = new Address();
"
" // Other codes go here.
" }
```

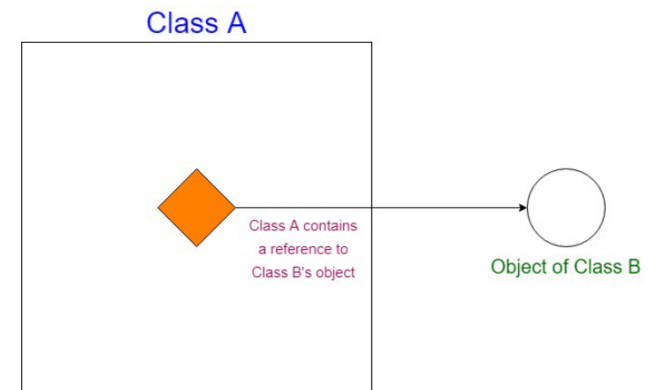
```
class Address
{
    .....
}
class Person
{
    Address addr = new Address();
    .....
    .....
}
```



The diagram illustrates a 'Has-A Relationship' between the `Person` and `Address` classes. A blue arrow points from the `Person` class to the `Address` class, with the text 'Has-A Relationship' written above it. This relationship is represented in the code by the `Person` class containing an instance of the `Address` class, `Address addr = new Address();`.

# Aggregation

- " Association (has a or part of) relationship, we are making a generic statement, which means that we are not concerned with the objects' lifetime dependency.
- " Aggregation uses **has-a relationship**
- " Java Aggregation allows only **one-to-one relationships**.
- " a class has a reference to an object o but does not control the lifetime of the object of the other class.
- " Ex: person and passport





# Example-Aggregation

```
class Country {  
    private String name;  
    private int population;  
    public Country(String n, int p) {  
        name = n;  
        population = p;  
    }  
    public String getName()  
    { return name; } }
```

```
class Person {  
    private String name;  
    private Country country;  
    // An instance of Country class public  
    Person(String n, Country c) { name = n;  
        country = c; }  
    public void printDetails() {  
        System.out.println("Name: " + name);  
        System.out.println("Country: " +  
            country.getName());  
    } }
```

```
class CountryAggregation {  
    public static void main(String args[])  
    {  
        Country country = new Country("USA", 1);  
        {  
            Person user = new Person("Donald Trump", country);  
            user.printDetails();  
        } // The user object's lifetime is over  
        System.out.println(country.getName()); // The country  
            object still exists!  
    }  
}
```

# Aggregation -Exercise

Create a class **Department** with the following private attributes,

Attribute	Datatype
departmentName	String
staff	Staff

Create a class **Staff** with the following private attributes,

Attribute	Datatype
staffName	String
designation	String

Include the following method in the **Department** class,

Method	Description
public displayStaff()	This method displays " <u>staffName</u> is working in the <u>departmentName</u> department as <b>designation</b> ".

Create a driver class **Main**, in the main method get the inputs from user, create the objects and call the methods.

# Composition

- " The practice of creating other class objects in your class is known as composition.
- " In this case, the class that creates the object of the other class is known as the owner, and it is responsible for the object's lifetime.
- " Composition relationships are **Part-of** relationships in which the part must be a component of the entire object.
- " In composition, the lifetime of the owned object is determined by the owner's lifetime.

# Example

```
class Engine {  
    private int capacity;  
    public Engine(int cap)  
    {    capacity = cap; }  
  
    public void engineDetails()  
    {  
        System.out.println("Engine  
details: " + capacity);  
    }  
}
```

```
class Tires {  
    private int noOfTires;  
    public Tires() {  
        noOfTires = 0;  
    }  
    public Tires(int nt) {  
        noOfTires = nt;  
    }  
    public void tireDetails()  
    {  
        System.out.println("Number  
of tyres: " + noOfTires);  
    }  
}
```

```
class Doors {  
    private int noOfDoors;  
  
    public Doors() {  
        noOfDoors = 0;  
    }  
  
    public Doors(int nod) {  
        noOfDoors = nod;  
    }  
  
    public void doorDetails() {  
        System.out.println("Number  
of Doors: " + noOfDoors);  
    }  
  
}
```

```
class Car {  
    private Engine eObj;  
    private Tires tObj;  
    private Doors dObj;  
    private String color;  
    public Car(String col, int cap, int nt, int nod)  
    {  
        this.eObj = new Engine(cap);  
        this.tObj = new Tires(nt);  
        this.dObj = new Doors(nod);  
        color = col;  
    }  
    public void carDetail()  
    {  
        eObj.engineDetails();  
        tObj.tireDetails();  
        dObj.doorDetails();  
        System.out.println("Car color: " + color);  
    }  
}
```

# Main

```
class CompositionCar {  
    public static void main(String[] args)  
    {  
        Car cObj = new Car("Black", 1600, 4, 4);  
        cObj.carDetail();  
    } }  

```

## UML NOTATION FOR CLASS RELATIONSHIPS

### Relationship

### UML Connector

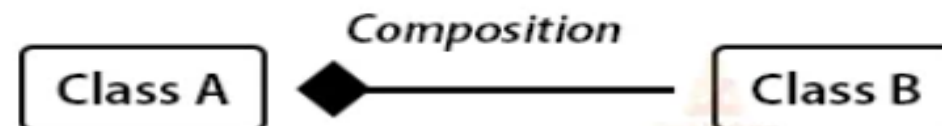
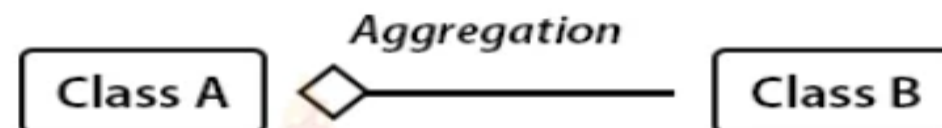
Inheritance



Interface Inheritance

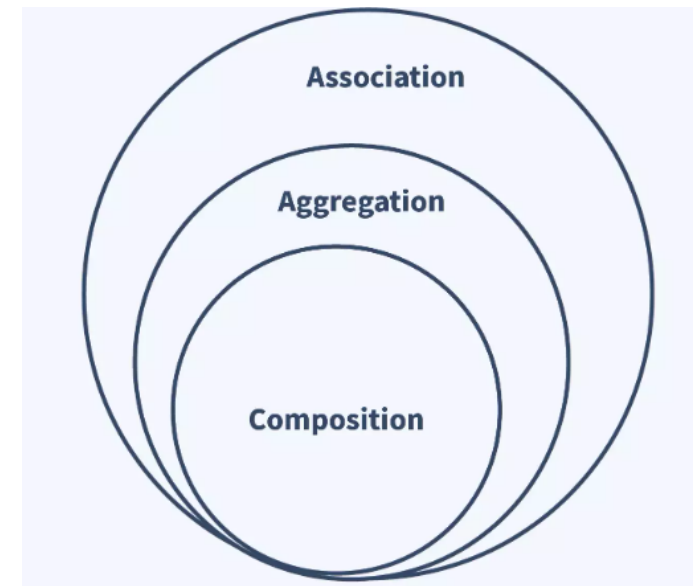


Dependency





Aggregation	Composition
Weak Association	Strong Association
Classes in relation can exist independently	One class is dependent on Another Independent class. The Dependent class cannot exist independently in the event of the non-existence of an independent class.
One class has-a relationship with another class	Once class belongs-to another class
Helps with code reusability. Since classes exist independently, associations can be reassigned or new associations created without any modifications to the existing class.	Code is not that reusable as the association is dependent. Such Associations once established will create a dependency, and these associations cannot be reassigned or new associations like aggregation, etc cannot be created without changing the existing class.



## A Superclass Variable Can Reference a Subclass Object

It is important to understand that it is the type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed.

# Constructors and Inheritance

```
class Base {  
Base()  
{  
    System.out.println("Base Class Constructor Called ");  
}  
}
```

```
class Derived extends Base {  
Derived() {  
    //super();  
    System.out.println("Derived Class Constructor Called ");  
}  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Derived d = new Derived();  
    }  
}
```

Base Class Constructor Called  
Derived Class Constructor Called

if the superclass has a constructor that requires any arguments (not ( )), you *must* put a constructor in the subclass and have it call the super-constructor (call to super-constructor must be the first statement)

```
class Base {
```

```
    int x;
```

```
    Base(int x) {
```

```
        x = x;
```

```
    }
```

```
}
```

```
public class Main {
```

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

```
        Derived d = new Derived(10, 20);
```

```
        d.Display();
```

```
    }
```

```
class Derived extends Base {
```

```
    int y;
```

```
    Derived(int x, int y) {
```

```
        super(x);
```

```
        y = y;
```

```
    }
```

```
    void Display() {
```

```
        System.out.println("x = "+x+", y = "+y);
```

```
    }
```

```
}
```

# Constructors and Initialization

---

- Classes use constructors to initialize instance variables
  - When a subclass object is created, its constructor is called.
  - It is the responsibility of the subclass constructor to invoke the appropriate superclass constructors so that the instance variables defined in the superclass are properly initialized
- Superclass constructors can be called using the "super" keyword
- It must be the first line of code in the constructor
- If a call to super is not made, the system will automatically attempt to invoke the no-argument constructor of the superclass.
  - Syntax: `Super(args)`

## Constructors - Example

---

```
class Room
{
    int length;
    int breadth;
    int result;
    Room(int x, int y)
    {
        length=x;
        breadth=y;
    }
    void area()
    {
        result =length*breadth;
    }
    void display()
    {
        System.out.println("area="+result);
    }
}
```



## Constructors - Example

---

```
class Bedroom extends Room
{
    int height;

    Bedroom( int x, int y, int z)
    {
        super(x,y);
        height=z;
    }
    int volume()
    {
        result=length*breadth*height;
    }
    void display()
    {
        System.out.println("volume="+result);
    }
}
```

# Upcasting and Downcasting

- There are two ways in which the objects can be initialized while inheriting the properties of the parent and child classes. They are:
- **Child c = new Child():** The use of this initialization is to access all the members present in both parent and child classes, as we are inheriting the properties.
- **Parent p = new Child():** This type of initialization is used to access only the members present in the parent class and the methods which are overridden in the child class.

# Example -upcasting

```
" class Father {  
"  
"   public void work()  
"   { System.out.println("Earning Father");}  
"  
"   }  
"  
" class Son extends Father {  
"  
"   public void play()  
"   { System.out.println("Enjoying son");}  
"  
"   }
```

```
class Main {  
    public static void main(String[] args)  
    {  
  
        Father father;  
        father = (Father) new Son();//upcasting  
        father.work();  
        // father.play(); //error  
    }  
}
```

# Example-downcasting

```
" class Main {  
"  
"    public static void main(String[] args)  
"  
"    {  
"  
"        Father father = new Son(); //Upcasting  
"  
"        Son son = (Son)father; //downcasting  
"  
"        son.work(); // works well  
"  
"        son.play(); // works well  
"  
"    }  
"  
" }
```

# Casting Objects

- " Converting a subclass type into a superclass type is called '**Generalization**' because we are making the subclass to become more general and its scope is widening. This is also called **widening or up casting**.
- " So, in widening or Generalization, **we can access all the superclass methods, but not the subclass methods**.
- " Converting a super class type into a sub class type is called '**Specialization**'. Here, we are coming down from more general form to a specific form and hence the scope is narrowed. Hence, this is called **narrowing or down-casting**.
  - When a superclass reference (referring to superclass object) is narrowed, then using that reference we can access neither methods of subclass nor methods of superclass.
  - When a subclass reference (referring to subclass object) is widened and then again narrowed, then using that reference we can access all the methods of the subclass as well as the superclass.

# Polymorphism

- „ Can treat an object of a subclass as an object of its superclass

A reference variable of a superclass type can point to an object of its subclass

A superclass reference can refer to an instance of the superclass OR an instance of ANY class which inherits from the superclass.

- „ These reference variables have many forms, that is, they are polymorphic reference variables
- „ They can refer to objects of their own class or to objects of the classes inherited from their class

# Polymorphism

---

- Polymorphism is: The method being invoked on an object is determined AT RUNTIME and is based on the type of the object receiving the message.
- Method Overriding forms the basis for java's powerful concepts: dynamic Method Dispatch.
- Dynamic Method Dispatch is the mechanism by which a call to an overridden function is resolved at run time, rather than compile time.
-



# Method LookUp

- " To determine if a method is legal the compiler looks in the class based on the declared type
  - if it finds it great, if not go to the super class and look there
  - continue until the method is found, or the Object class is reached and the method was never found. (Compile error)
- " To determine which method is actually executed the run time system
  - starts with the actual **run time class** of the object that is calling the method
  - search the class for that method
  - if found, execute it, otherwise go to the super class and keep looking
  - repeat until a version is found

# Dynamic method Dispatch

# Method Overriding

---

- Subclasses inherit all methods from their superclass
  - Sometimes, the implementation of the method in the superclass does not provide the functionality required by the subclass.
  - In these cases, the method must be overridden.
  - . Dynamic method dispatch is the mechanism by which a call to an overridden function is resolved at run time, rather than compile time.

- To override a method, provide an implementation in the subclass.

Derived/sub classes defining methods with same name, return type and arguments as those in the parent/super class, *override* their parents methods:

- The method in the subclass **MUST** have the exact same signature as the method it is overriding.
- It must have the same name ,return type and arguments as that of the super class.
- Only non static members can be overridden

- " When an overridden method is called through a superclass reference, 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.
- " Thus, this determination is made at run time.
- " When different types of objects are referred to, different versions of an overridden method will be called.
- " In other words, 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.
- " Therefore, 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.

## Method Overriding

- " If subclass has a method of a superclass (same signature), that method overrides the superclass method:

```
public class A { ...  
    public int M (float f, String s) { bodyA }  
}
```

```
public class B extends A { ...  
    public int M (float f, String s) { bodyB }  
}
```

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

```
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 object A
        r.callme();    // call A's version of callme

        r=b;          // r refers to a B object
        r.callme();    //call B's version of callme

        r=c;          //r refers to an object C
        r.callme();    // calls C's version of callme  }
    }
}
```



# 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 Methods and Final Classes

---

- Methods can be qualified with the final modifier
  - Final methods cannot be overridden.
  - This can be useful for security purposes.

```
public final boolean validatePassword(String username, String Password)
{
    [...]
}
```

- Classes can be qualified with the final modifier
  - The class cannot be extended
  - This can be used to improve performance. Because there can be no subclasses, there will be no polymorphic overhead at runtime.

```
public final class Color
{
    [...]
}
```

# Overriding Methods

```
class A {  
    int j = 1;  
    int f( ) { return j; }  
}
```

```
class B extends A {  
    int j = 2;  
    int f( ) {  
        return j; }  
}
```

# Overriding Methods

```
class override_test {  
    public static void main(String args[]) {  
        B b = new B();  
        System.out.println(b.j);    // refers to B.j prints 2  
        System.out.println(b.f());  // refers to B.f prints 2  
  
        A a = (A) b;  
        System.out.println(a.j);    // now refers to a.j prints 1  
        System.out.println(a.f());  // overridden method still refers to B.f() prints 2 !  
    }  
}
```

**Object Type Casting**



```
>java override_test  
2  
2  
1  
2
```

# FINALIZE

- " **Finalize :**
- " It is Obvious that a constructor method is used to initialize an object when it is declared. This process is known as initialization.
- " Java supports a **concept** called finalization, which is just opposite to initialization.
- " We know the java run-time system is an automatic garbage collecting system. It frees up the memory resources used by the objects.
- " But object may hold other non-object resources such as file descriptors or window system fonts.
- " The garbage collector cannot free these resources. In order to free these resources we must use a finalize(r) method. This is similar to destructors in c++.
- " The finalize(r) method is simply finalize() and can be added to any class. The finalize method should explicitly define the tasks to be performed.

## **Abstract Method and Classes :**

We can indicate that a method must always be redefined in a subclass, thus making overriding compulsory.

This is done using the modifier keyword `abstract` in the method definition .

When a class contains atleast one abstract method then that class should also be defined as `abstract`.

We cannot use abstract classes to instantiate objects directly. For example.

```
Pgcourse pg=new Pgcourse();
```

Is illegal because Pgcourse is an abstract class.

The abstract methods of an abstract class must be defined in its subclass.

We cannot declare abstract constructors or abstract static methods.

**Abstract** class PgCourse

{

float inpassmin=12.5;

float extpassmin=37.5;

**abstract** void Test();

**abstract** void report();

void cut-off()

{

int cut\_mark=40;

System.out.println(“The cut-off value is”+cut\_mark);

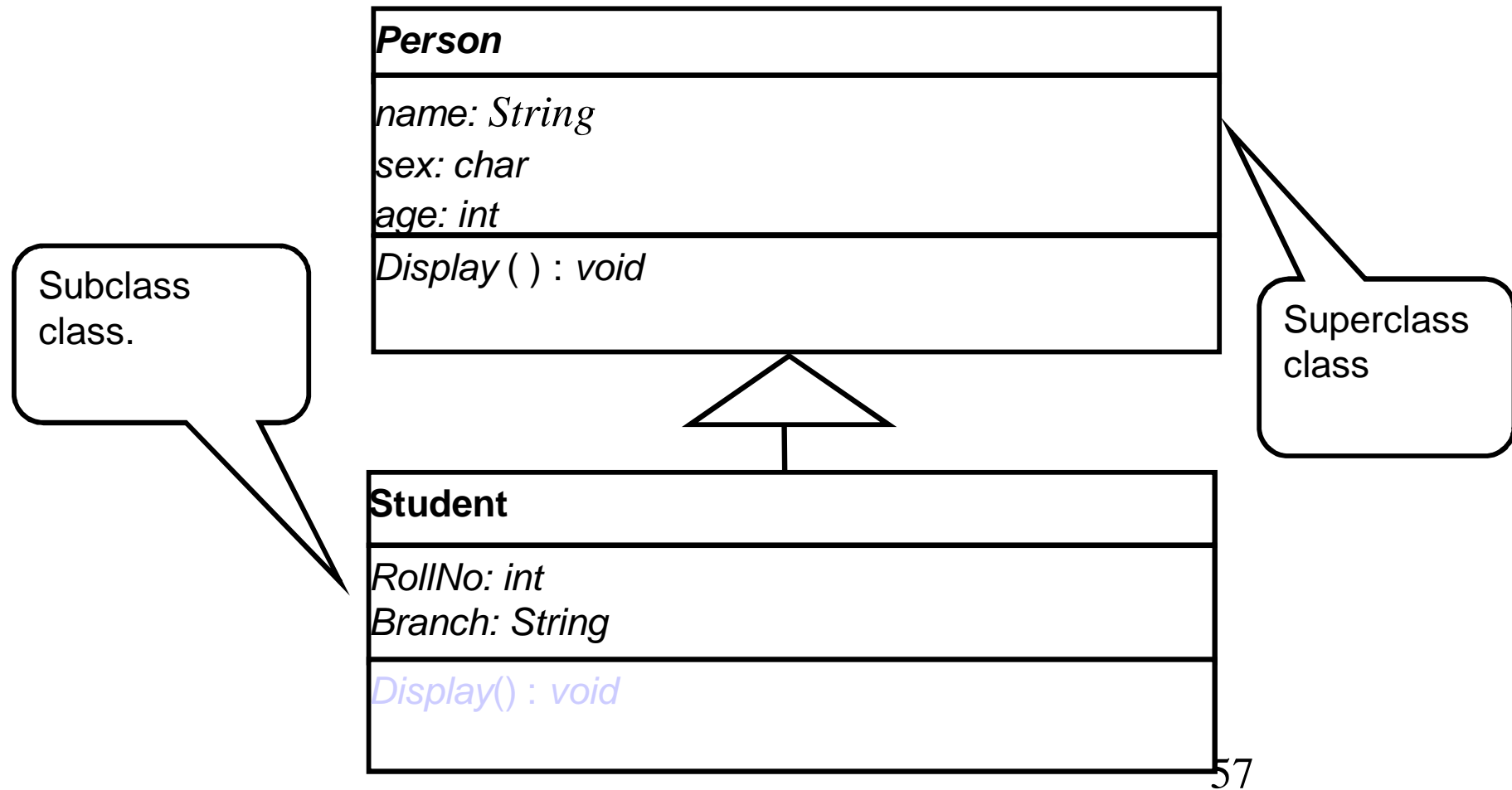
}

}

```
class Mca extends Pgcourse
{
    void Test()
    {
        //Here, Method body should be fully defined,
        //Otherwise declare this method also as abstract
    }
    void report()
    {
        //Define the method body of this method too
    }
    //we can also override cut-off() even though it is already defined
}
```



# Using All in One: Person and Student



# Person class: Parent class

// Student.java: Student inheriting properties of person class

```
class person
{
    private String name;
    protected char sex; // note protected
    public int age;
    person()
    {
        name = null;
        sex = 'U'; // unknown
        age = 0;
    }
}
```

**person(String name, char sex, int age)**

**{**

**this.name = name;**

**this.sex = sex;**

**this.age = age;**

**}**

**String getName()**

**{          return name;**

**}**

**void Display()**

**{**

**System.out.println("Name = "+name);**

**System.out.println("Sex = "+sex);**

**System.out.println("Age = "+age);**

**}**

**}**

# Student class: Derived class

class student extends person

```
{    private int RollNo;

    String branch;

    student(String name, char sex, int age, int RollNo, String branch)

    {    super(name, sex, age); // calls parent class's constructor with 3 arguments

        this.RollNo = RollNo;

        this.branch = branch;

    }

    void Display() // Method Overriding

    {

        System.out.println("Roll No = "+RollNo);

        System.out.println("Name = "+getName());

        System.out.println("Sex = "+sex);

        System.out.println("Age = "+age);

        System.out.println("Branch = "+branch);

    }
```

```

class MyTest
{
    public static void main(String args[] )
    {
        student s1 = new student("Rama", 'M', 21, 1, "Computer Science");
        student s2 = new student("Sita", 'F', 19, 2, "Software Engineering");

        System.out.println("Student 1 Details...");
        s1.Display();
        System.out.println("Student 2 Details...");
        s2.Display();

        person p1 = new person("Rao", 'M', 45);
        System.out.println("Person Details...");
        p1.Display();

    }
}

```

# Output

[raj@mundroo] inheritance [1:154] java MyTest

Student 1 Details...

Roll No = 1

Name = Rama

Sex = M

Age = 21

Branch = Computer Science

Student 2 Details...

Roll No = 2

Name = Sita

Sex = F

Age = 19

Branch = Software Engineering

Person Details...

Name = Rao

Sex = M

# 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

# Constructor Chaining

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Performs Faculty's tasks");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Performs Employee's tasks ");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

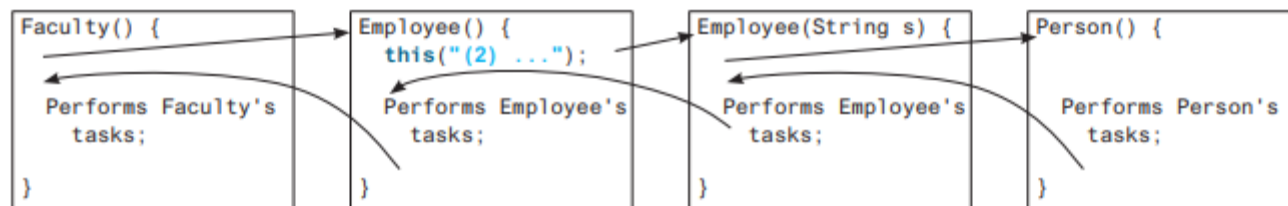
class Person {
    public Person() {
        System.out.println("(1) Performs Person's tasks");
    }
}
```





- (1) Performs Person's tasks
- (2) Invoke Employee's overloaded constructor
- (3) Performs Employee's tasks
- (4) Performs Faculty's tasks

The program produces the preceding output. Why? Let us discuss the reason. In line 3, `new Faculty()` invokes `Faculty`'s no-arg constructor. Since `Faculty` is a subclass of `Employee`, `Employee`'s no-arg constructor is invoked before any statements in `Faculty`'s constructor are executed. `Employee`'s no-arg constructor invokes `Employee`'s second constructor (line 13). Since `Employee` is a subclass of `Person`, `Person`'s no-arg constructor is invoked before any statements in `Employee`'s second constructor are executed. This process is illustrated in the following figure.



# Copy Constructor in Java

- Copy constructors create a new object by using the values of an existing object.
- These constructors can create shallow as well as deep clones.

# Copy Constructor for Classes with Referenced Types

- " If the class fields are primitives or immutables, then shallow cloning is the same as deep cloning.
- " `Student(Student s)`
- " `{ this.name = s.getName();`
- " `this.gpa = s.getGpa(); //creating a new address object`  
`this.studentAddress = new`  
`Address(s.getStudentAddress().postalCode); }`

# Example

```
public class Ninja{  
    private String name;  
    private int hitPoints;  
    private Weapon weapon; //Dynamically Allocated Object  
    private Belt Tools [];
```

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

```
        Ninja Gero = new Ninja("George", 42, Sword, belt[lightning,stars, knife]);  
    }
```

```
    " Ninja Gracie=new Ninja("George"); //CC Call
```

```
Ninja Ninja(Ninja copy){ Gracie
```

```
newNinja.name = copy.name;
```

```
newNinja.hitPoints = copy.hitPoints;
```

```
newNinja.weapons = copy.Weapons;
```

```
newNinja.belt = copy.belt;
```

```
return newNinja;
```

```
}
```

```
Ninja Ninja(Ninja newNinja){  
;  
    newNinja.name = copy.name;  
    newNinja.hitPoints = copy.hitPoints;  
    newNinja.weapons = new weapons(copy.Weapons);  
    //deep copy  
    newNinja.belt = new belt();  
    //deep copy  
    for( Tool tool: copy.belt){  
        newNinja.belt.append(new Tool(tool));  
    }  
    return newNinja;  
}
```