
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS390 Foundamental Programming Practices (FPP)

Lecture 5: Inheritance, Interfaces, and Polymorphism

Life is structures in Layers

Wholeness of the Lesson

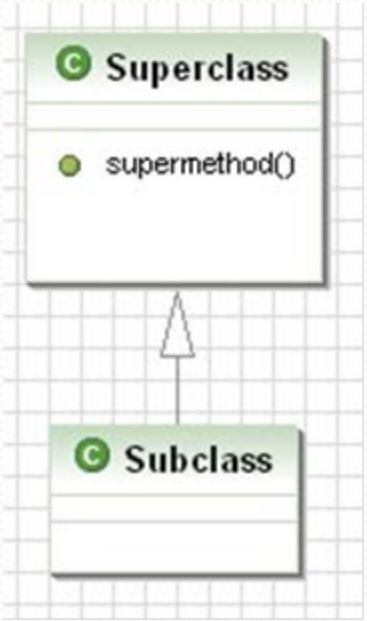
Java supports inheritance between classes in support of the OO concepts of inherited types and polymorphism. Interfaces support encapsulation, play a role similar to abstract classes, and provide a safe alternative to multiple inheritance. ***Likewise, relationships of any kind that are grounded on the deeper values at the source of the individuals involved result in fuller creativity of expression with fewer mistakes.***

Outline of Topics

- Introduction to Inheritance – Example of Subclassing a Class
- The "IS-A" and LSP Criteria for Proper Use of Inheritance
- Access Modifier - protected
- Rules for Subclass Constructors
- Inheritance and the Object Class
- Introduction to Polymorphism
- Order of Execution with Inheritance
- Abstract Class
- Introduction to Java Interfaces, `Comparable`, Functional Interfaces
- New Java 8 Features for Interfaces
- Introduction to the Reflection Library
- The `Object` Class
  - The `toString` Method
  - The `equals` Method
  - The `hashCode` Method
  - The `clone` Method: Shallow and Deep Copies

Introduction to Inheritance

- *Definition.* A class Subclass *inherits from* another class Superclass if objects of type Subclass have automatic access to the "available" methods and variables that have been defined in class Superclass. By "automatic access" we mean that no explicit instantiation of (or reference to) the class Superclass is necessary in order for objects of type Subclass to be able to call methods defined in class Superclass. By "available" methods and variables, we mean methods and variables that have been declared either *public* or *protected* (or have *package level access* if in the same package).




```
class Superclass {
   protected void supermethod() {
         int x = 0;
   }
}

class Subclass extends Superclass {
}

class Main {
   public static void main(String[] args)
   {
         Subclass sub = new Subclass();
         sub.supermethod();
   }
}
```

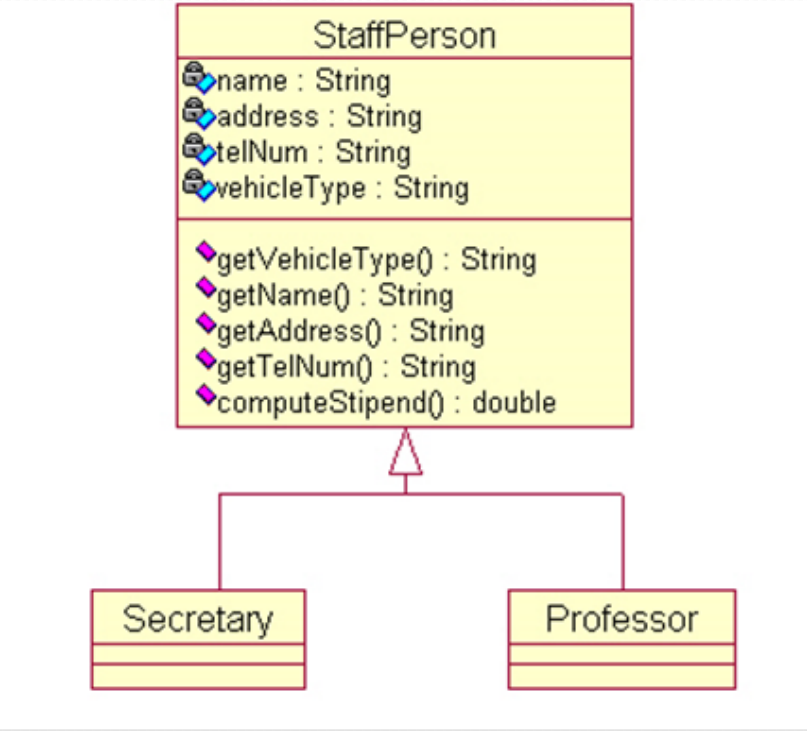
A class, method, variable labeled *protected* is accessible to all *subclasses*.

- *Motivation.* In our programming projects, we may find that we define two classes that have many of the same fields and methods. It is natural to think of a single class that generalizes the two classes and that contains the code needed by both.

Secretary
  properties:
    name
    address
    phone number
    drivesVehicle
    salary
  behavior:
    computeSalary()

Professor
  properties:
    name
    address
    phone number
    drivesVehicle
    salary
  behavior:
    computeSalary()

Strategy: Create a *generalization* of `Secretary` and `Professor` from which both of these classes *inherit*. A `StaffPerson` class can be defined having all four fields and related methods, and `Secretary` and `Professor` can be defined so they are *subclasses* of `StaffPerson`.



When the classes have this relationship, we may view the type of an instance of a subclass as being that of the superclass. For example, we can instantiate like this:

```
StaffPerson person1 = new Professor():
StaffPerson person2 = new Secretary();
```

This is similar in spirit to the automatic conversions that are done for primitive types:

```
byte b = 8;
int k = b;
```

An Example of Superclass and Subclass: Manager subclass of Employee

```
//Employee class, as defined in previous lessons
class Employee {
    Employee(String aName, double aSalary, int aYear, int aMonth, int aDay) {
        name = aName;
        salary = aSalary;
        hireDay = LocalDate.of(aYear,aMonth,aDay);
    }

    // instance methods
    public String getName() {
        return name;
    }
    public double getSalary() {
        return salary;
    }
    public LocalDate getHireDay() {
        return hireDay;
    }
    public void raiseSalary(double byPercent) {
        double raise = salary * byPercent / 100;
        salary += raise;
    }

    //instance fields
    private String name;
    private double salary;
    private LocalDate hireDay;
}
```

```
class Manager extends Employee {
  public Manager(String name, double salary, int year,
        int month, int day) {
    super(name,salary,year,month,day);
    bonus = 0;
  }
  @Override
  public double getSalary() {
    //no direct access to private variables of
    //superclass
    double baseSalary = super.getSalary();
    return baseSalary + bonus;
  }
  public void setBonus(double b) {
    bonus = b;
  }
  private double bonus;
}
```

```
class ManagerTest {
   public static void main(String[] args) {
     Manager boss = new Manager("Boss Guy", 80000,1987, 12, 15);
     boss.setBonus(5000);

     Employee[] staff = new Employee[3];

       staff[0] = boss;
     staff[1] = new Employee("Jimbo", 50000, 1989, 10, 1);
     staff[2] = new Employee("Tommy", 40000, 1990, 3,15);

     //print names and salaries
     for(Employee e : staff) {
       System.out.println( "name: " + e.getName() +
                           "salary: " + e.getSalary());
     }
   }
}
```

Points to observe:

- `Manager` provides all the "services" of `Employee`, with additional functionality (involving bonuses) and overriding functionality (`getSalary` method) – so it's a good candidate for *extending* `Employee`.
- We use the *extends* keyword to indicate that `Manager` is a *subclass* of `Employee`
- A `Manager` instance can freely use the `getName` and `getHireDay` methods of its superclass `Employee` – no need to re-code these methods. However, special methods that are unique to `Manager` (in particular, the `setBonus` method) cannot be called on an `Employee` instance.

```
Manager m = new Manager(. . .);
m.getName();        //ok
m.setBonus(5000);          //ok

Employee e = new Employee(. . .);
e.getName();               //ok
e.setBonus(5000);  //compiler error
```

- We *override* the `getSalary` method in the `Manager` class.
  - This means that the method is defined differently from its original version in `Employee`. A `Manager` object computes `salary` differently from `Employee` objects.
- Still wish to *use* `getSalary` in `Employee` , but add the value of `bonus` to it. How can this be done?
  - In general, how to access the *superclass version* of a method from within a *subclass*?

  Solution: Use **`super`** to indicate that you are accessing the superclass version.

Best Practice. Use the @Override annotation on `getSalary`.
Two reasons :

- It is possible for another user of your code not to realize that your method overrides a method in a superclass.

- Provides a compiler check that your method *really is* overriding a superclass method.

- In the `Manager` constructor, we wish to *reuse* the constructor that is found in `Employee`, but we also want to include more code. This is accomplished by using the **super** keyword again (but it has a different meaning here).

  Like **this** in constructor, the use of **super** must occur on the first line of the constructor body.

- ***Polymorphic types***. The `0`th element of the `staff` array was defined to be of type `Manager`, yet we placed it in an array of `Employee` objects. The fact that an object variable can refer to an object of a given type as well as objects that belong to subtypes of the given type is called *polymorphism*.

- ***Dynamic binding.*** When the `getSalary` method is called on `staff[0]`, the version of `getSalary` that is used is the version that is found *in the* `Manager` *class*. This is possible because the JVM keeps track of the actual type of the object when it was created (that type is set with execution of the "new" operator). The correct method body (the version that is in Manager) is associated with the getSalary method at runtime – this "binding" of method body to method name is called *late binding* or *dynamic binding*.

Polymorphism

- Polymorphism refers to the ability of an object to take on many forms.
- *A variable of a supertype can refer to a subtype object.*
- **Compile time Polymorphism(Early/static Binding) :**
  - **Method overloading**
- `obj.doIt(); obj.doIt(10); Obj.doIt(10,20);`
- This line of code might execute different methods at different times, if the object that `obj` points to changes.
- **Runtime Polymorphisrm(Late/dynamic binding) :**
  - **Method Overriding** :Polymorphic references are resolved at run time this is called *dynamic binding.*

Rules for overriding

1. The method must be apply to an instance method. Overriding does not apply to static method.
2. The overriding method must have the same name as the overridden method.
3. The overriding method must have the same number of parameters of the same type in the same order as the overridden method
4. Return type of the overriding and overridden methods must be the same
5. The access level of the overriding method must be at least the same or more relaxed than that of the overridden method.

Overridden(Parent)	**Overriding Access level (Child)**
public	public
protected	public, protected
package-level	public, protected, package-level

Is it overriding?

```
public class S{
public void print(){
  System.out.println(“S”);
        }
}
public class T extends S{
public void print(String msg){
  System.out.println(msg);
        }
}
```

Overloading vs. Overriding

- Overridden methods are in different classes related by inheritance; overloaded methods can be in the same class.
- Overridden methods have the same signature and return type; overloaded methods have the same name but a different parameter list, return types do not play any roles in overloading.
- Overriding applies only to instance methods; Any method(static/non-static) can be overloaded.
- Overloading is determined during the compile time but overriding determined during the runtime.

Inheritance and Access

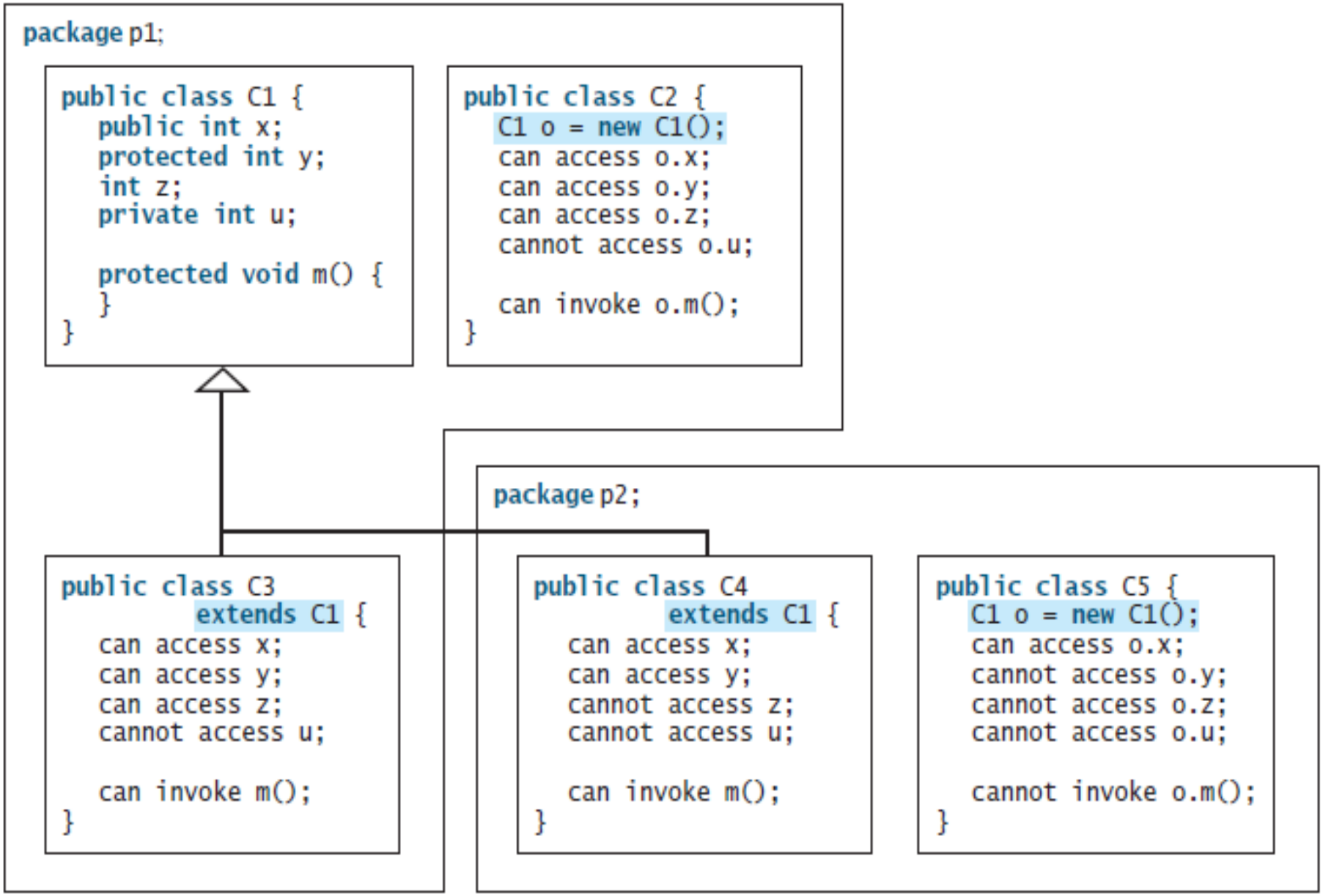
Base class access	Accessibility in derived class
`public`	`Yes`
`protected`	`Yes`
`private`	Inaccessible
Unspecified (package access-default)	`Yes`

Access Modifier

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with `public` visibility are inherited; those with `private` visibility are not
- But `public` variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

The protected Modifier

- The `protected` modifier allows a member of a base class to be inherited into a child
- Protected visibility provides more encapsulation than public visibility does
- However, protected visibility is not as tightly encapsulated as private visibility
- Try the implementation of next slide to understand modifiers.



Visibility modifiers are used to control how data and methods are accessed.

Main Point

One class (the *subclass)* inherits from another class (the *superclass*) if all proteced and public data and methods in the superclass are automatically accessible to the subclass, even though the subclass may have additional methods and data not found in the superclass. Java supports this notion of inheritance. In Java syntax, a class is declared to be a subclass of another by using the *extends* keyword. Likewise, individual intelligence "inherits from" cosmic intelligence, though each "implementation" is unique.

Correct Use of Inheritance

Here are two tests to check whether one class should inherit from another.

- `Manager` IS-A `Employee` – it's not just that the two classes have some methods in common, but a manager really is an employee. This helps to verify that inheritance is the right relationship between these classes.

- *Liskov Substitution Principle (LSP).* Another test is: Can a `Manager` instance be used whenever an `Employee` instance is expected? The answer is yes, since every manager really is an employee, and partakes of all the properties and behavior of an employee, though managers support extra behavior.

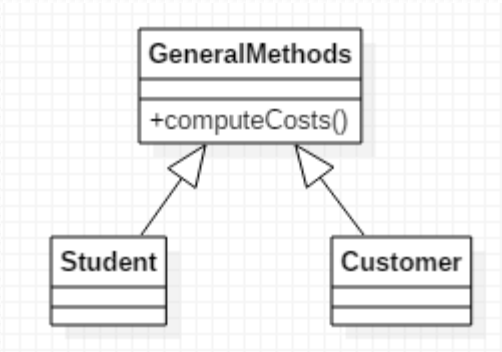
Common Mistake: Indiscriminate Generalization

- The following strategy is a mistake: Place methods common to several classes into one superclass for all of them. Then all the classes have immediate access to methods that they all can use.

  This is undesirable because, eventually, some methods and variables in the superclass will not be relevant for some of the subclasses – those subclasses will therefore offer "services" that they cannot possibly provide.

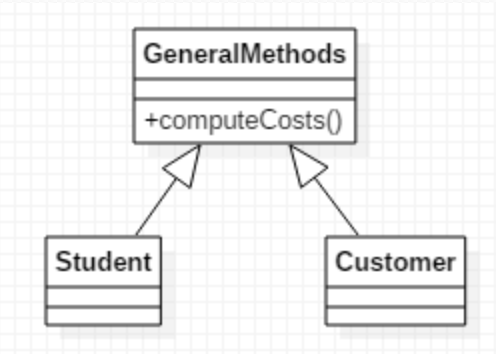
(continued)

- At first, this may seem reasonable

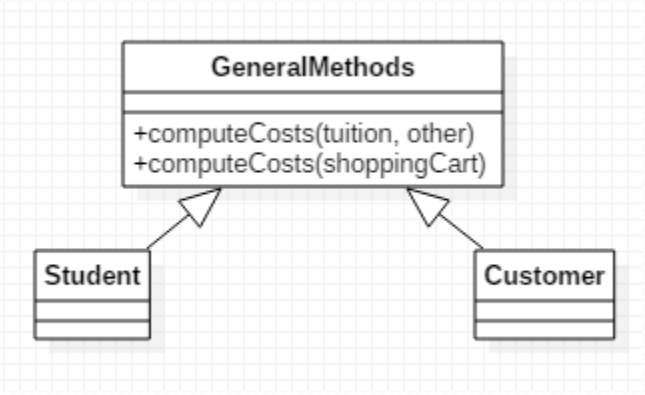


(continued)

- At first, this may seem reasonable



- As your project evolves, you may find that different versions of computeCosts are needed for Student and Customer



Now a `Customer` seems to be supporting `computeCosts` with input `tuition` and `other`. This undermines the purpose of the `Customer` class

(continued)

- Often, common methods can be placed in a *utility class*.

```
public class Student {
    void aMethod() {
        double tuition = 5000;
        double other = 3000;
        //...
        double val = Util.computeCosts(tuition, other);
        //...
    }
}
```

```
public class Customer {
    void aMethod() {
        ShoppingCart cart = new ShoppingCart();
        //...populate cart
        double val = Util.computeCosts(cart);
        //...
    }
}
```

```
public class Util {
    private Util() {
        //private constructor
    }
    public static double computeCosts(double tuition, double other) {
        double cost = 0.0;
        //. . .compute
        return cost;
    }

    public static double computeCosts(ShoppingCart cart) {
        double cost = 0.0;
        //unpack cart and  compute
        return cost;
    }

}
```

Main Point

As a matter of good design, a class C should not be made a subclass of a class D unless C "IS-A" D. Likewise, individual intelligence "is" cosmic intelligence, though this relationship requires time to be recognized as true.

Rules for Subclass Constructors

The Rule:

a subclass constructor must make use of one of the constructors from its superclass

Reason for the rule The state of the superclass (values of its instance variables) should be set *by the superclass* (not by the subclass). so during construction, the subclass must request the superclass to first set its state, and then the subclass may perform further initialization of its own state.

Example. `Employee/Manager`:

```
class Employee{
    Employee(String name, double salary, int y, int m, int d){
        //…//
    }
}
class Manager extends Employee {
    Manager(String name, double salary, int y, int m, int d) {
        super(name,salary,y,m,d); //makes use of superclass constructor
    }
}
```

Note: It is not necessary for any of the subclass constructors to have the same signature as any of the superclass constructors. However, each of the subclass constructors must access one of the superclass constructors in its implementation.

```
class Employee{
   Employee(String name, double salary, int y,
            int m, int d){
      //…//
   }
}
class Manager extends Employee {
   Manager(String name, double salary,
           double bonus,//different but ok
            int y, int m, int d) {
      super(name,salary,y,m,d);
      this.bonus = bonus;
   }
}
```

The subclass may make use of the implicit default constructor *only if* either

A. the no-argument constructor of the superclass has been explicitly defined, OR
B. no constructor in the superclass is explicitly defined

In either of these cases, the subclass may make use (possibly implicitly) of the superclass' s default constructor.

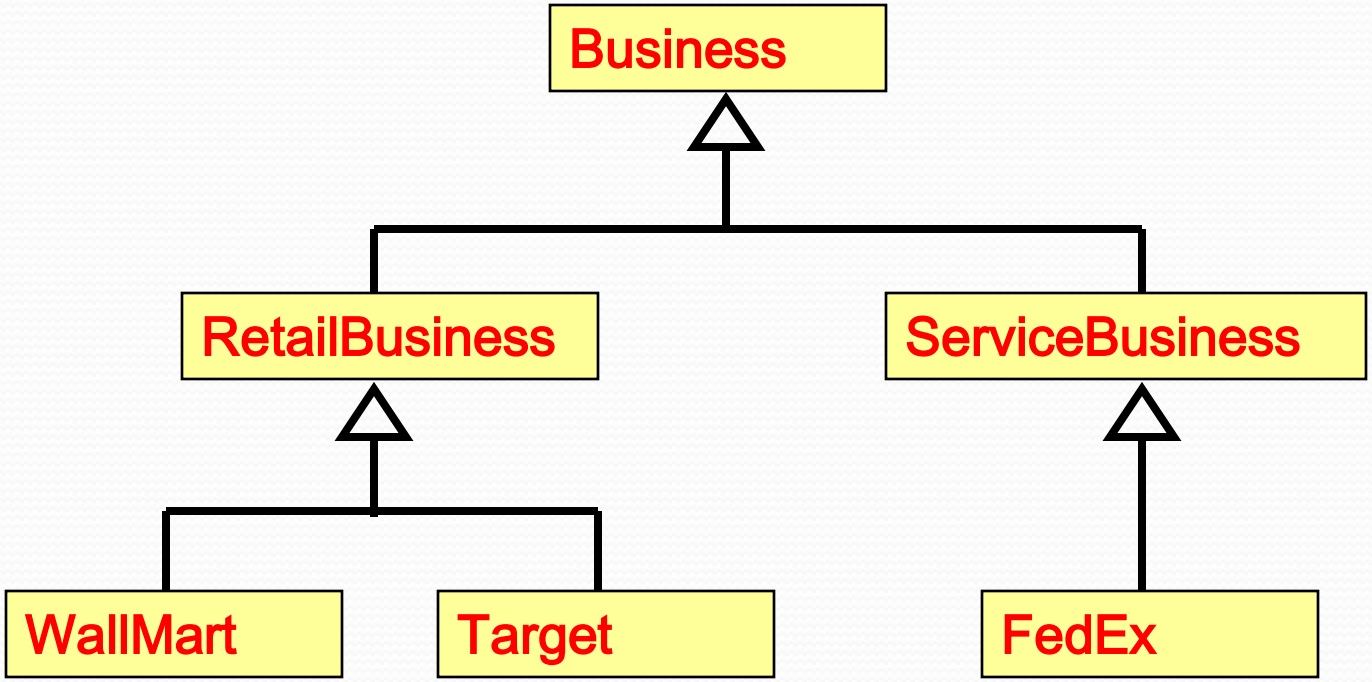
```
//Case A.
class Employee{
        Employee(String name, double salary, int y, int m, int d){
                //…//
        }
        //explicit coding of default constructor since another
        //constructor is present
        Employee() {
                //…//
        }
}
class Manager extends Employee {
        //no explicit constructor call here, so the superclass
        //default constructor is used implicitly
}
```

```
//Case B.

  class Employee{
       //...//
  }
  class Manager extends Employee {
       //...//
  }
```

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a *class hierarchy*.



Multiple Inheritance

- Java supports *single inheritance*, meaning that a derived class can have only one parent class
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Java does not support multiple inheritance
- In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead

Inheritance and the Object Class

- In Java, there is a class called `Object`. Every class created in Java (either in the Java libraries, or user-defined) belongs to the inheritance hierarchy of `Object`.

  For example:

```
class MyClass {

}
```

  This `MyClass` class automatically inherits from `Object`, even though we do not write syntax that declares this fact.

  In later slides, we will discuss the (primarily public) methods that belong to `Object`, and that are therefore inherited by every class in Java.

- *Using the* `instanceof` *operator to check type.*

The following code returns true:

```
"Hello" instanceof java.lang.String
```

In general, you can query Java about the type of any runtime object by using `instanceof`. The general synatx is

```
ob instanceof <classname>
```

where `ob` is of type `Object` (or any subtype). This expression will return true if the *runtime type* of `ob` really is of the specified type, or if the class of `ob` is a subclass of (or a subclass of a subclass of...etc) the specified type. Therefore, for example, if `e` is an instance of `Employee` and s is a `String`, both of the following are true

```
e instanceof Object
s instanceof Object
```

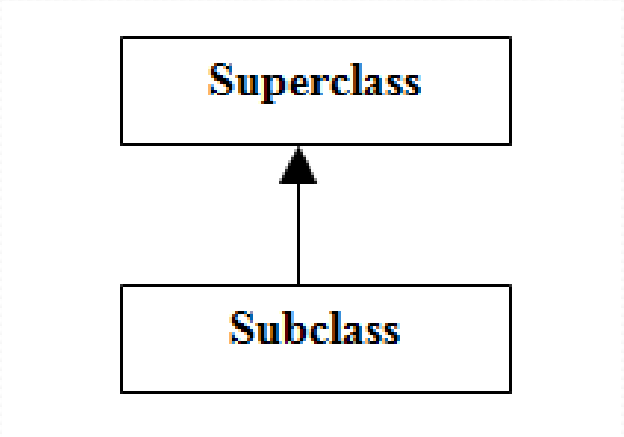
Whenever the `instanceof` operator returns `true`, the `object` on the left side of the expression can be viewed as having type indicated on the right side (via polymorphic type assignment). So in this example, we could type `e` and `s` above as `Objects`:

```
Object ob_e = e;
Object ob_s = s;
```

instaceof and casting

```
Object[] stuff = {"Java", 10.11,12,13,16.11,20,"Hi"};
double sum = 0;
for(int i=0;i<stuff.length;i++){
if(stuff[i] instanceof Number) // checking instance
{
Number next = (Number)stuff[i]; // Down casting
sum+=next.doubleValue();
}
}
System.out.println("Sum of Doubles = " + sum);
```

Order of Execution with Inheritance



Suppose, as in a typical case, we have `Subclass` as a subclass of `Superclass`. When we run

```
new Subclass()
```

the sections of the code are executed according to the following scheme:

- In `Superclass`, all static variables are initialized and all static initialization blocks are run, in the order in which they appear in the file.
- In `Subclass`, all static variables are initialized and static initialization blocks are run, in the order in which they appear in the file.
- In `Superclass`, all instance variables are initialized and all object initialization blocks are run, in the order in which they appear in the file
- In `Superclass`, the (relevant) constructor is run.
- In `Subclass`, all instance variables are initialized and all object initialization blocks are run, in the order in which they appear in the file
- In `Subclass`, the (relevant) constructor is run.

[See Demo – `ClassE.java`]

Demo Code

- ManagerTest1.java // Employee Manager Example used on Slide for more concepts
- SuperThisDemo.java
- DynamicBind.java
- InstanceofDemo.java
- Hierarchical.java
- ClassE.java // Order of execution