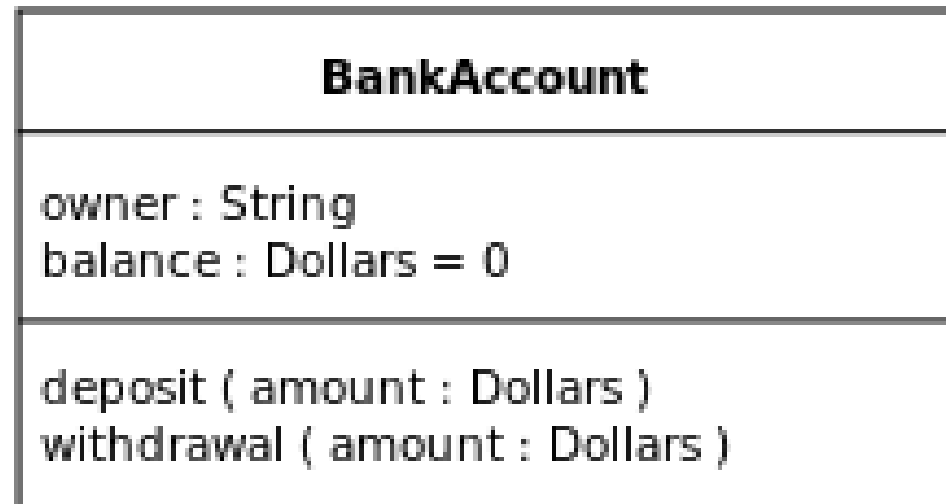


# Java OOP\_1

- Association
  - Aggregation
  - Composition
  - Dependency
  - Realization
  - Generalization
- Inheritance
- Polymorphism

# Class diagram

- The class diagram is the main building block of [object oriented](#) modelling.
- In the diagram, classes are represented with boxes which contain three parts:
  - The top part contains the name of the class. It is printed in bold and centered, and the first letter is capitalized.
  - The middle part contains the attributes of the class. They are left-aligned and the first letter is lowercase.
  - The bottom part contains the methods the class can execute. They are also left-aligned and the first letter is lowercase.
- In the design of a system, a number of classes are identified and grouped together in a class diagram which helps to determine the static relations between those objects.



# Class diagram : Visibility

## Visibility

To specify the visibility of a class member (i.e., any attribute or method), these notations must be placed before the member's name.

+	Public
-	Private
#	Protected
/	Derived (can be combined with one of the others)
~	Package

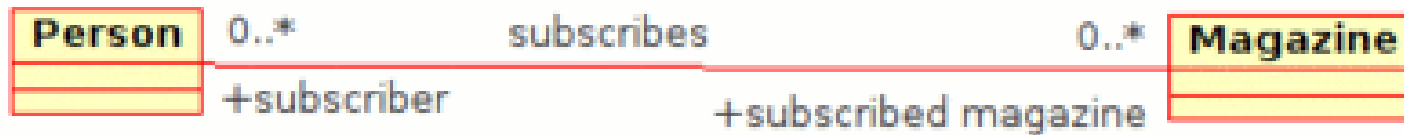
To indicate a *classifier scope* ( *static*) for a member, its name must be underlined. Otherwise, instance scope is assumed by default.

# Class diagram : Association

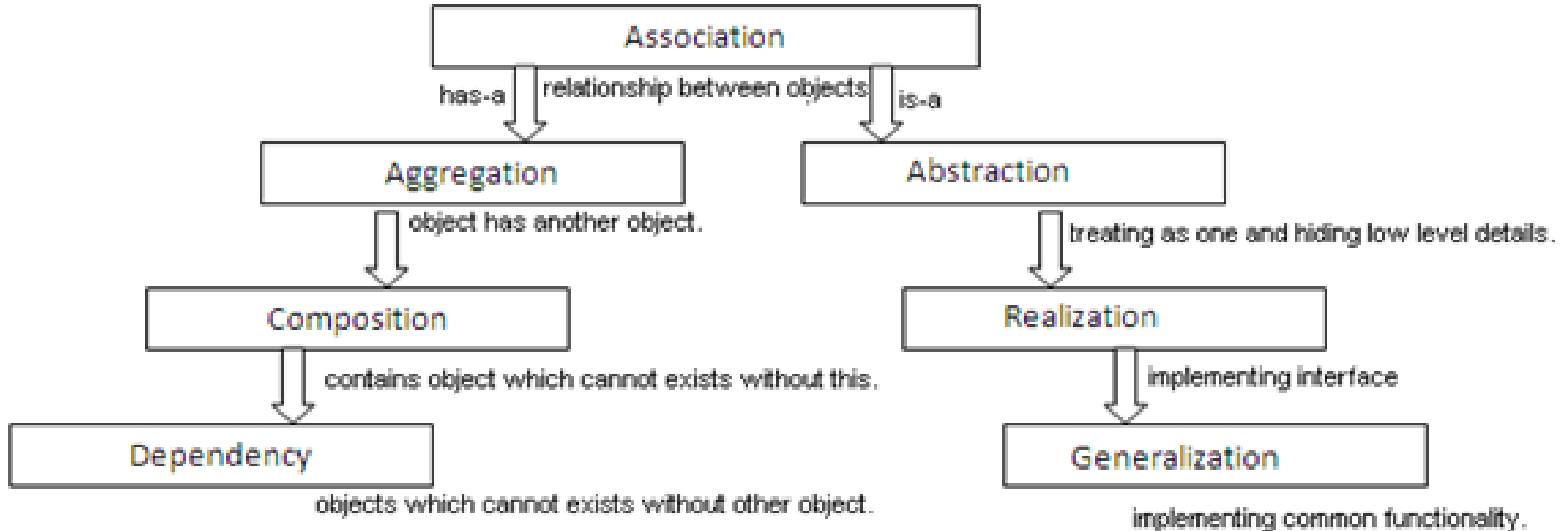
## Relationships

A relationship is a general term covering the specific types of logical connections found on class and object diagrams.

- An association represents a family of links. A binary association (with two ends) is normally represented as a line. An association can link any number of classes.
- Association represents the relationship shared among the objects of two classes

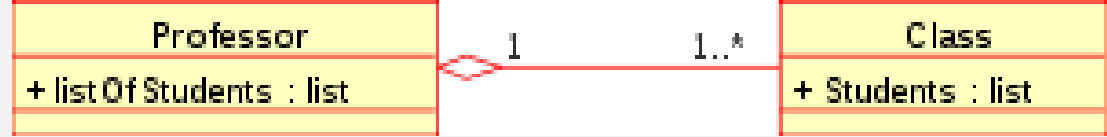


# Association, Aggregation, Composition & Dependency



# Class diagram : Aggregation

## Aggregation

- [Aggregation](#) is a variant of the "has a" association relationship; aggregation is more specific than association.
- As shown in the image, a Professor 'has a' class to teach.

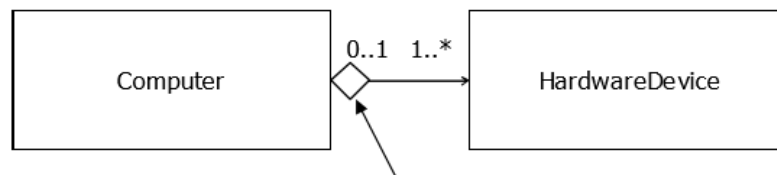
```
classDiagram
    Professor "1" o-- "1..*" Class
    Professor "+ listOfStudents : list"
    Class "+ Students : list"
```

The diagram illustrates an aggregation relationship between a Professor and a Class. The Professor class has an attribute + listOfStudents : list. The Class class has an attribute + Students : list. The association is represented by a red line with a hollow diamond at the Professor end, indicating that a Professor can have one or more classes (1 to 1..\*).
- *Aggregation* can occur when a class is a collection or container of other classes, but the contained classes do not have a strong *lifecycle dependency* on the container. The contents of the container are not automatically destroyed when the container is.
- In [UML](#), it is graphically represented as a *hollow diamond shape* on the containing class with a single line that connects it to the contained class
- Example :Library has a Students and Books. Here the student can exist without library, the relation between student and library is aggregation.

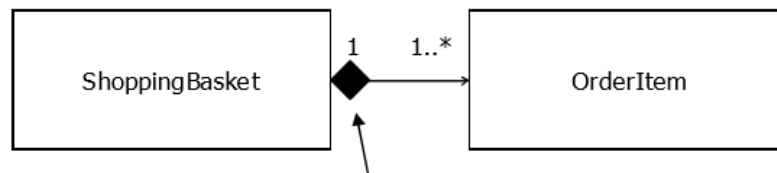
# Class diagram : Composition

- Composition is a stronger variant of the "has a" association relationship; composition is more specific than aggregation.
- *Composition* usually has a strong *lifecycle dependency* between instances of the container class and instances of the contained class(s): if the container instance is destroyed, normally every instance that it contains is destroyed as well.
- The UML graphical representation of a composition relationship is a *filled* diamond shape on the containing class end of the tree of lines that connect contained class(es) to the containing class.

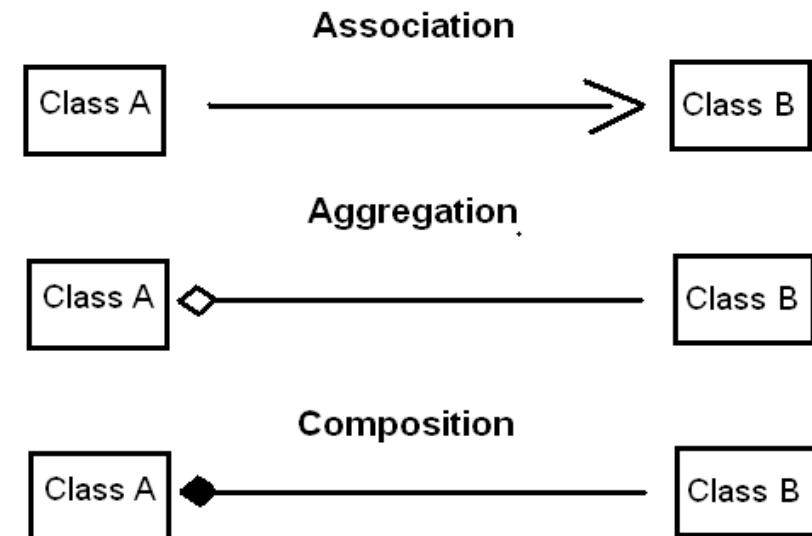
## Aggregation & Composition



Aggregation – is made up of objects that can be shared or exchanged



Composition – is composed of objects that cannot be shared or exchanged and live only as long as the composite object



# Class diagram : Multiplicity

## Multiplicity

This association relationship indicates that (at least) one of the two related classes make reference to the other. This relationship is usually described as "A has a B" (a mother cat has kittens, kittens have a mother cat).

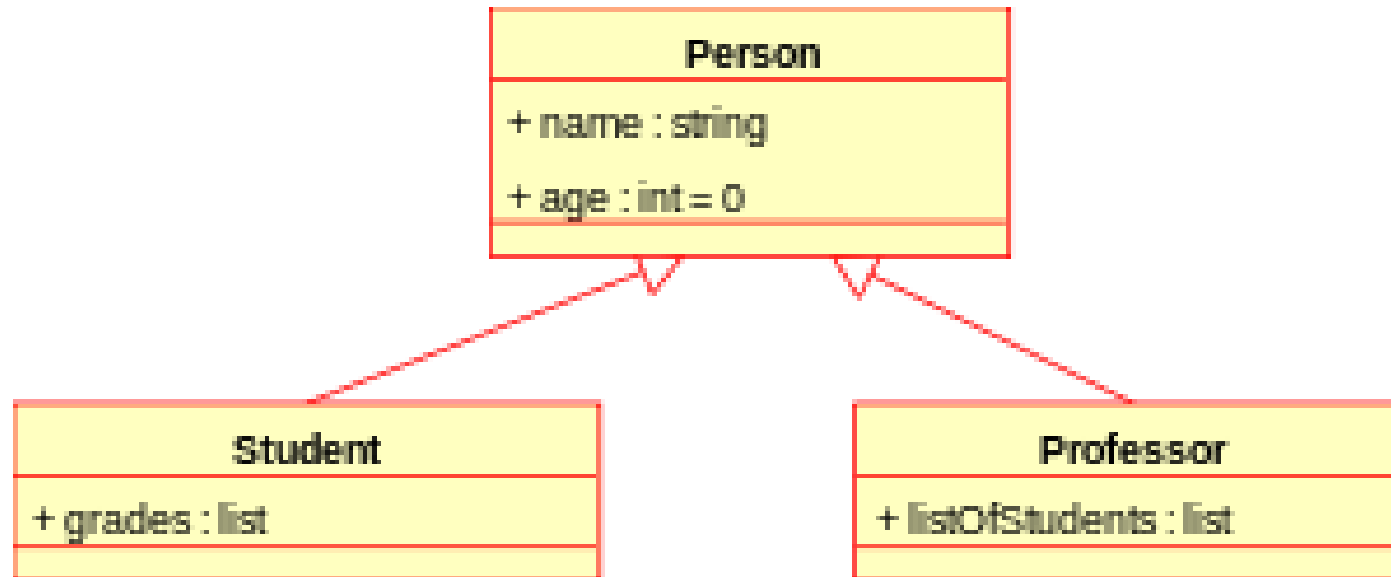
The UML representation of an association is a line with an optional arrowhead indicating the *role* of the object(s) in the relationship, and an optional notation at each end indicating the *multiplicity* of instances of that entity (the number of objects that participate in the association).

<b>0..1</b>	No instances, or one instance
<b>1</b>	Exactly one instance
<b>0..*</b>	Zero or more instances
<b>1..*</b>	One or more instances



# Class diagram : Generalization

- Indicates that one of the two related classes (the *subclass*) is considered to be a specialized form of the other (the *super type*) and the superclass is considered a '**Generalization**' of the subclass.
- In practice, this means that any instance of the subtype is also an instance of the superclass.
- The UML graphical representation of a Generalization is a hollow [triangle](#) shape on the superclass end of the line (or tree of lines) that connects it to one or more subtypes.
- The generalization relationship is also known as the [inheritance](#) or "*is a*" relationship.

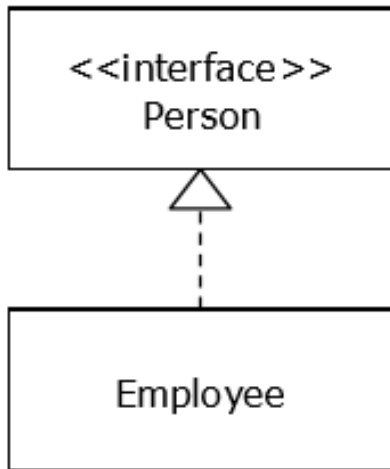


# Class diagram : Realization

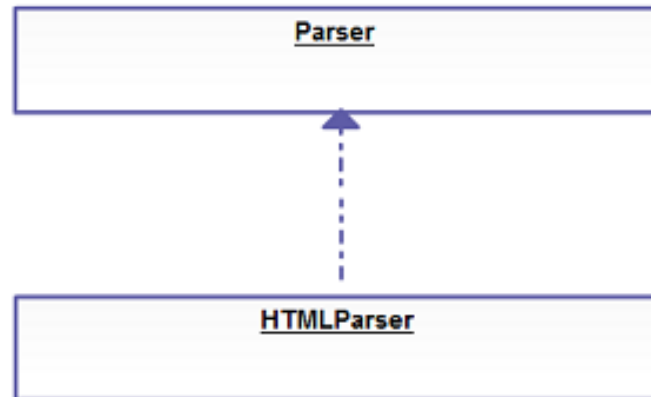
## Realization

In UML modelling, a realization relationship is a relationship between two model elements, in which one model element (the client) realizes (**implements** or executes) the behavior that the other model element (the supplier) specifies.

The UML graphical representation of a Realization is a hollow triangle shape on the interface end of the *dashed* line (or tree of lines) that connects it to one or more implementers



In a realization relationship, one entity (normally an interface) defines a set of functionalities as a contract and the other entity (normally a class) “realizes” the contract by implementing the functionality defined in the contract.



# Class diagram : Dependency

## Dependency

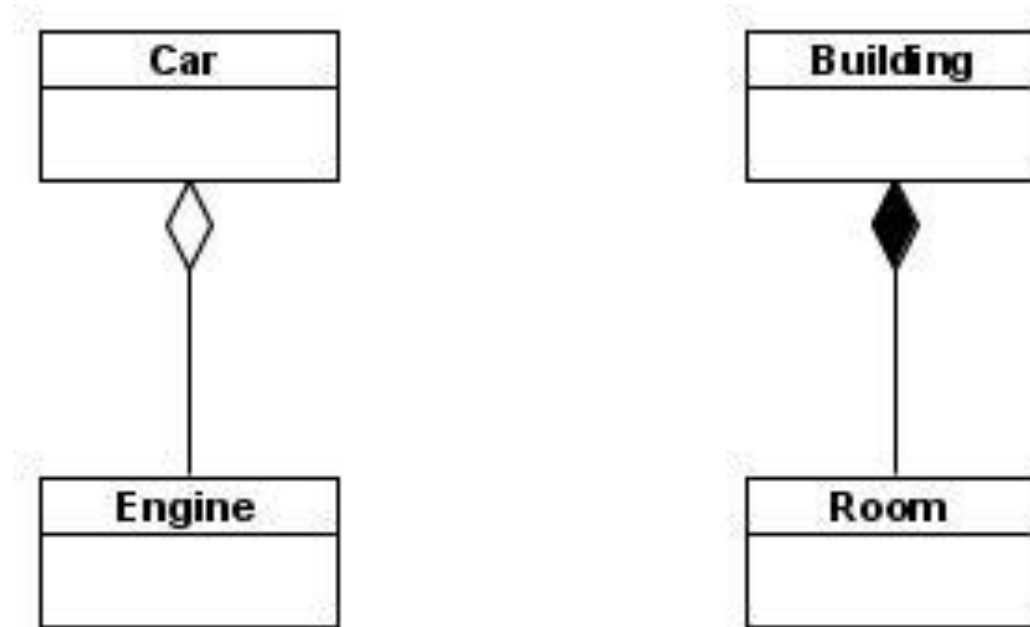
- This is a weaker form of bond which indicates that one class depends on another because it **uses** it at some point in time.
- One class depends on another if the independent class is a parameter variable or local variable of a method of the dependent class.
- Sometimes the relationship between two classes is very weak. They are not implemented with member variables at all. Rather they might be implemented as method arguments.
- For example, A **person** uses a **credit** card, an **employee** uses a **swipe** card etc.

# Aggregation and Composition

## Aggregation and Composition

- Both aggregation and composition represent a whole-part (has-a/part-of) association.
- The main differentiator between aggregation and composition is the lifecycle dependence between whole and part.
- In aggregation, the part may have an independent lifecycle, it can exist independently. When the whole is destroyed the part may continue to exist.
- For example, a car has many parts. A part can be removed from one car and installed into a different car. If we consider a salvage business, before a car is destroyed, they remove all saleable parts. Those parts will continue to exist after the car is destroyed.
- Composition is a stronger form of aggregation. The lifecycle of the part is strongly dependent on the lifecycle of the whole. When the whole is destroyed, the part is destroyed too.
- For example, a building has rooms. A room can exist only as part of a building. The room cannot be removed from one building and attached to a different one. When the building ceases to exist so do all rooms that are part of it.

# Aggregation and Composition



The relationship between *contained instance (part-of)* and *container class(whole)* is called as **has-a relationship**.

# Implementation in Java

In Java the difference between aggregation and composition boils down to the following: is the variable that holds the part object accessible to objects other than the whole?

If the answer is yes, then we have aggregation, if the answer is no then we have composition.

In order to enforce a composition relationship we need to make sure that the part object is accessible only by the whole object, which means that:

- The part object must be created inside the whole object (either inside constructor, as part of a static block or some init method which is invoked when the whole is created.)
- The instance variable that holds the part must be private
- There is no getter/setter for accessing the part (there is no way an outside object has access to the part object)

# Aggregation - Java sample code

```
public class Car {  
    private String make;  
    private int year;  
    private Engine engine;  
  
    public Car(String make, int year, Engine engine) {  
        this.make = make;  
        this.year = year;  
        this.engine = engine;  
    }  
  
    public String getMake() {  
        return make;  
    }  
  
    public int getYear() {  
        return year;  
    }  
  
    public Engine getEngine() {  
        return engine;  
    }  
}
```

*The engine object is created outside and is passed as argument to Car constructor when this Car object is destroyed, the engine is still available to objects other than Car.*

*If the instance of Car is garbage collected the associated instance of Engine may not be garbage collected (if it is still referenced by other objects)*

Java Project : JavaAggregationComposition

Package: com.capgemini.business-tier

Class Name: Car

Class Name: Engine

Package: com.capgemini.presentation-tier

Class Name: CarTester

# Aggregation - Java sample code

```
public class Engine {  
    private int engineCapacity;  
    private int engineSerialNumber;  
  
    public Engine(int engineCapacity, int engineSerialNumber) {  
        this.engineCapacity = engineCapacity;  
        this.engineSerialNumber = engineSerialNumber;  
    }  
  
    public int getEngineCapacity() {  
        return engineCapacity;  
    }  
  
    public int getEngineSerialNumber() {  
        return engineSerialNumber;  
    }  
}
```



# Composition- Java sample code

```
public class Car {  
    private String make;  
    private int year;  
    private Engine engine;  
  
    public Car(String make, int year, int engineCapacity, int engineSerialNumber) {  
        this.make=make;  
        this.year=year;  
        engine = new Engine(engineCapacity, engineSerialNumber);  
    }  
  
    public String getMake() {  
        return make;  
    }  
  
    public int getYear() {  
        return year;  
    }  
  
    public int getEngineSerialNumber() {  
        return engine.getEngineSerialNumber();  
    }  
  
    public int getEngineCapacity() {  
        return engine.getEngineCapacity();  
    }  
}
```

We create the engine using parameters passed in Car constructor  
only the Car instance has access to the engine instance.  
When Car instance is garbage collected, the engine instance is garbage collected too.

# Dependency

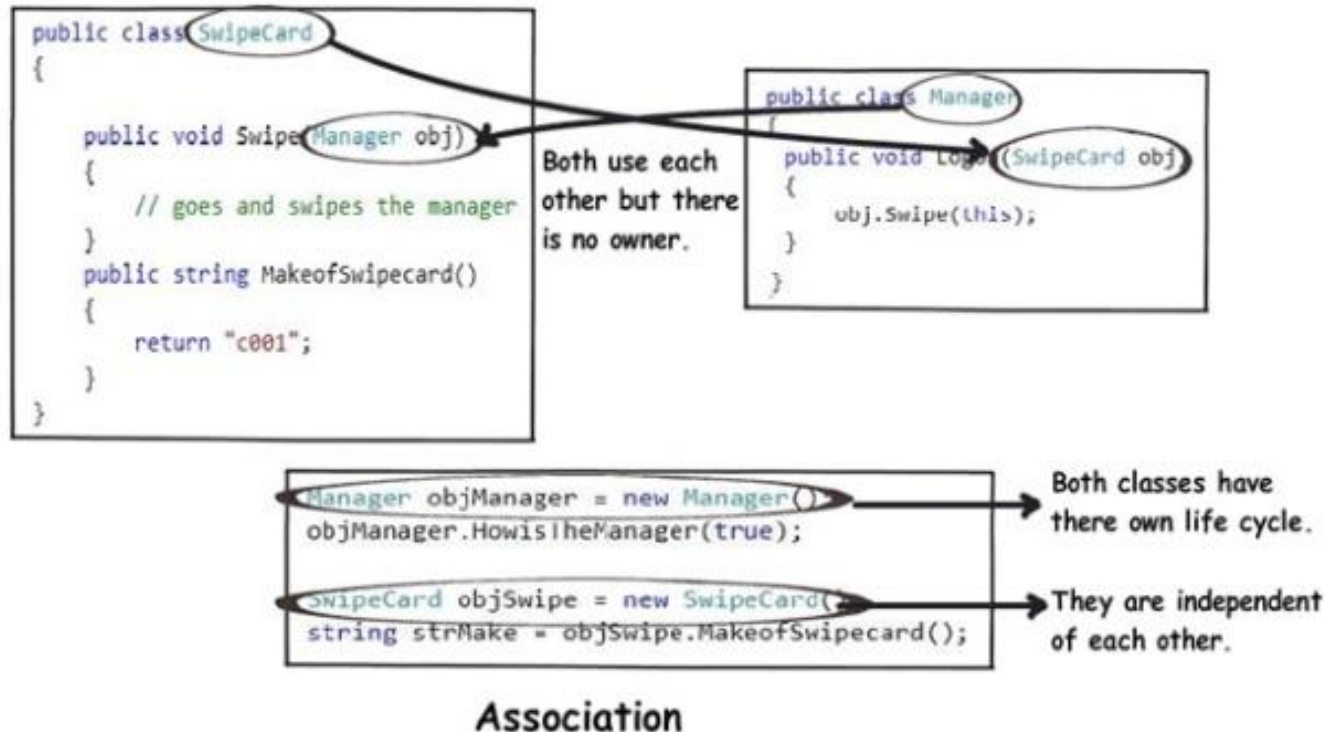
## Dependency: uses-a relationship

Uses-a relationship is one in which a method of one class uses a object of another class and vice-versa to perform the task.

Manager **uses a** swipe card to enter XYZ premises

In this requirement, the manager object and the swipe card object use each other but they have their own object life time.

In other words, they can exist without each other. **The most important point in this relationship is that there is no single owner.**



The above diagram shows how the **SwipeCard** class uses the **Manager** class and the **Manager** class uses the **SwipeCard** class.

You can also see how we can create objects of the **Manager** class and **SwipeCard** class independently and they can have their own object life time.

This relationship is called the "Dependency" relationship.

Inheritance (is-a relationship)

# Generalization

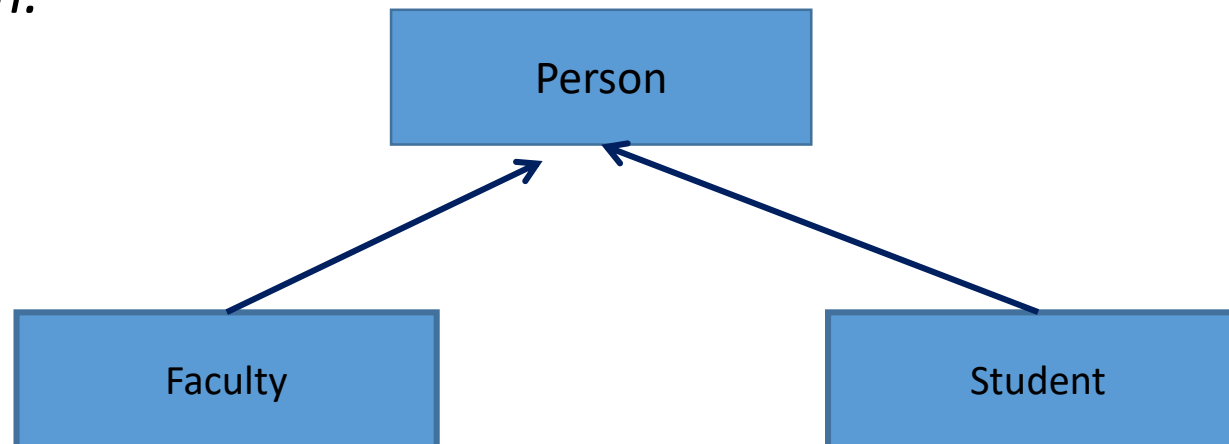
## Generalization

The Generalization relationship ("is a") indicates that one class (*subclass*) is considered to be a specialized form of another class (super class).

Super class is considered as '*Generalization*' of subclass.

The generalization relationship is also known as *inheritance* or "*is a*" relationship.

*Example:* Consider there exists a class named Person. A student *is a* person. A faculty *is a* person. Therefore here the relationship between student and person, similarly faculty and person is *generalization*.



# Inheritance

- Inheritance is process of creating a new class from an existing class.
- The parent class is termed **super** class and the inherited class is the **sub** class
- The keyword **extends** is used by the sub class to inherit the features of super class
- Inheritance leads to **reusability** of code

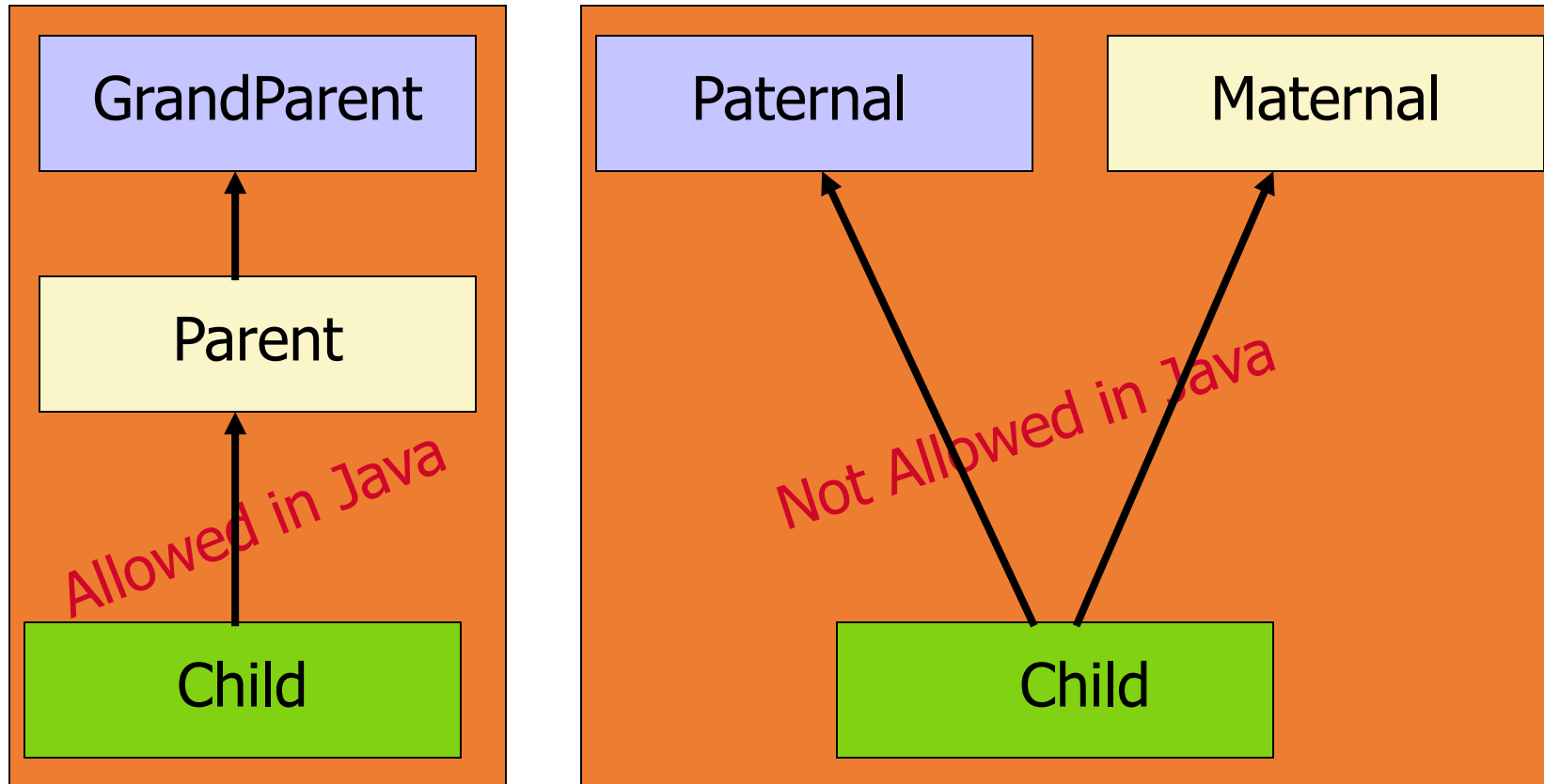
```
class Person {  
    /*attributes and functionality of Person  
    defined*/  
}  
  
class Student extends Person {  
    /*inherits the attributes and  
    functionalities  
    of Person and in addition add its own  
    specialties*/  
}
```

Person is a generalized class

Student "is a" Person and is a specialized class

# Inheritance

- **Multi-level** inheritance is allowed in Java but not **multiple** inheritance



- **Multiple** inheritance can be managed in Java using interfaces

# Inheritance

## What You Can Do in a Subclass ?

- A subclass inherits all of the *instance* members of its parent, no matter what package the subclass is in.
- The public, package-private (default) and protected members of super class are visible in its subclass.
- Private members of a super class though inherited in its subclass but not visible in the subclass.
- The inherited fields can be used directly, just like any other fields.
- You can declare a *instance field* in the subclass with the same name as the one in the super class, thus *hiding* it (not recommended) , called as **data hiding**.
- You can declare new fields in the subclass that are not in the super class.
- The inherited methods can be used directly as they are.

# Inheritance

- You can write a new *instance method* in the subclass that has the same signature as the one in the super class (including its return type) , thus *overriding* it, called as **method overriding**.
- You can write a new *static method* in the subclass that has the same signature as the one in the super class, thus *hiding* it , called as **method hiding**.
- You can declare new methods in the subclass that are not in the super class.
- A subclass constructor method **implicitly invokes** the default constructor of the super class, by placing *super()* method as its first statement.
- **Overloaded constructors are not inherited.**
- You can write a subclass constructor that invokes the parameterized constructor methods of the super class explicitly by using the keyword `super`

## ***Private Members in a Super class***

The instance of subclass has memory allocated to the private members of its parent class, but are not visible nor directly accessible. However, if the super class has public or protected methods for accessing its private fields, then these can also be used by the subclass.



# private fields of super class

## Java Project: JavaInheritanceProject

```
package com.deloitte.trg.service;
public class A {
    private int x=10;
    int y=20;
    static int z = 100;

    //instance method
    public int getX(){
        return x;
    }

    //static method
    public static int getZ(){
        return z;
    }

    //instance method
    public void showData(){
        System.out.println(x+", "+y+", "+z);
    }
}
```

10  
20  
30  
100  
200  
100  
200

```
package com.deloitte.trg.service;
```

```
public class B extends A{
    int y=30;
    static int z=200;
```

```
    public static int getZ(){
        return z;
    }
```

```
    public void showData(){
        System.out.println(x); Error since x is private field
        System.out.println( getX());
        System.out.println(super.y);
        System.out.println(this.y);
        System.out.println(A.z);
        System.out.println(B.z); // or simply z
        System.out.println(A.getZ());
        System.out.println(B.getZ());
    }
}
```

data hiding

method hiding

method overriding

Accessing the  
hidden field of a  
super class

```
package com.deloitte.trg.ui;
public class Tester {
    public static void main(String [] args){
        new B().showData()
    }
}
```

# protected fields of super class

## Java Project: JavaInheritanceProject

```
package com.deloitte.trg.service;  
public class A1{  
    protected int x=10;  
    protected int y=20;  
    protected static int z = 100;
```

```
    //instance method  
    public int getX(){  
        return x;  
    }  
}
```

```
    //static method  
    public static int getZ(){  
        return z;  
    }  
}
```

```
    //instance method  
    public void showData(){  
        System.out.println(x+", "+y+", "+z);  
    }  
}
```

```
package com.deloitte.trg.service.new;
```

```
public class B1 extends A1{  
    int y=30; // data hiding  
    static int z=200;
```

```
    // method hiding  
    public static int getZ(){return z;}
```

```
    // method overriding  
    public void showData(){  
        System.out.println(x);  
        System.out.println( getX());  
        System.out.println(super.y);  
        System.out.println(y);  
        System.out.println(A1.z);  
        System.out.println(B1.z);  
        System.out.println(A1.getZ());  
        System.out.println(B1.getZ());  
    }
```

```
10  
10  
20  
30  
100  
200  
100  
200
```

protected fields of super class are accessible in the sub class even if it is in another package.

```
package com.deloitte.trg.ui;  
public class Tester {  
    public static void main(String [] args){  
        new B1().showData()  
    }  
}
```

- Using *super*
- Runtime Polymorphism in Java
  - Method Overriding

# Initializing super class constructors in the subclass constructors

For proper initialization of the data fields, Java must ensure that every constructor method of subclass calls its super class constructor method.

If we do not explicitly invoke super class constructor within the subclass constructor, **Java implicitly calls the default super class constructor** . i.e. it implicitly inserts `super();` statement as its first statement

*Syntax of invoking default super class constructor :*

*syntax:*

**`super();`**

To explicitly invoke super class overloaded constructor methods

*Parameterized super class constructor,*

*Syntax:*

**`super(<arguments>);`**

Note: `super()` should be the **first statement** within the subclass constructor method.

# Constructor Chaining

Constructor calls are chained; when an object is created, a sequence of constructor methods are invoked.

Because a super class constructor is always invoked as the first statement of its subclass constructor, the body of the **Object( parent class for all the classes)** constructor always runs first, followed by the constructor of its subclass and on down the class hierarchy to the class that is being instantiated.

**In short, execution of the constructors is done from top to bottom.**

```
Class A {  
    A(){  
        System.out.println("I'm class A");  
    }  
}  
Class B extends A {  
    B(){  
        System.out.println("I'm class B");  
    }  
}  
Class C extends B {  
    C(){  
        System.out.println("I'm class C");  
    }  
}
```

```
new C();
```

*Output :*

I'm class A

I'm class B

I'm class C

# Inheritance Example

## Java Project: JavaInheritanceProject

```
package com.deloitte.trg.service;

public class A1{
    private int a;
    public A1(int a){
        this.a=a;
    }
    public int getA(){
        return a;
    }
}
```

```
public A1(){
    System.out.println("Hello");
}
```

```
package com.capgemini.bussinesstier;

public class B1 extends A1{
    private int b;
    public B1(){
        super(); //optional
        System.out.println("Welcome");
    }
    public B1(int a,int b) {
```

Compilation error since there is no default constructor in super class

**super(a); // explicitly invoking 1-arg constructor of super class**

```
        this.b=b;
    }
    public void show(){
        System.out.println( getA()+", "+b);
    }
}
```

```
package com.deloitte.trg.service;

public class InheritanceTester
    public static void main(String [] args){
        new B1();

        B1 bObj = new B1(5,10);
        bObj.show();
    }
}
```

Hello  
Welcome  
0,10

Hello  
Welcome  
5,10

# Inheritance Example

If sub class constructor does not explicitly call the super class constructor, then JVM implicitly invokes default constructor of Super class.

```
class A{  
protected int a;
```

```
A(){a=0;}
```

```
A(int a){  
    this.a=a;
```

```
}  
}
```

```
class B extends A{  
private int b;
```

```
B(){super();}
```

```
B(int a,int b){  
    this.b=b;  
}
```

```
B(int a,int b) {  
    super(a); // invoking 1-arg constructor of super class  
    this.b=b;  
}
```

```
void show(){  
    System.out.println("a = " + a + ", b = "+b);  
}
```

**Java Project: JavaInheritanceProject**

```
B bobj = new B(5,10)
```

```
bobj.show();
```

Output:

a = 0 , b = 10

Output : a= 5 , b = 10

# METHOD OVERRIDING

- ***Redefining*** a super class instance method in a sub class is called method overriding. **In method overriding, method signature includes return type**, unlike method overloading.
- *Some rules in overriding*
  - The method signature i.e. method name, parameter list and return type have to match exactly
  - The overridden method can widen the accessibility but not narrow it.



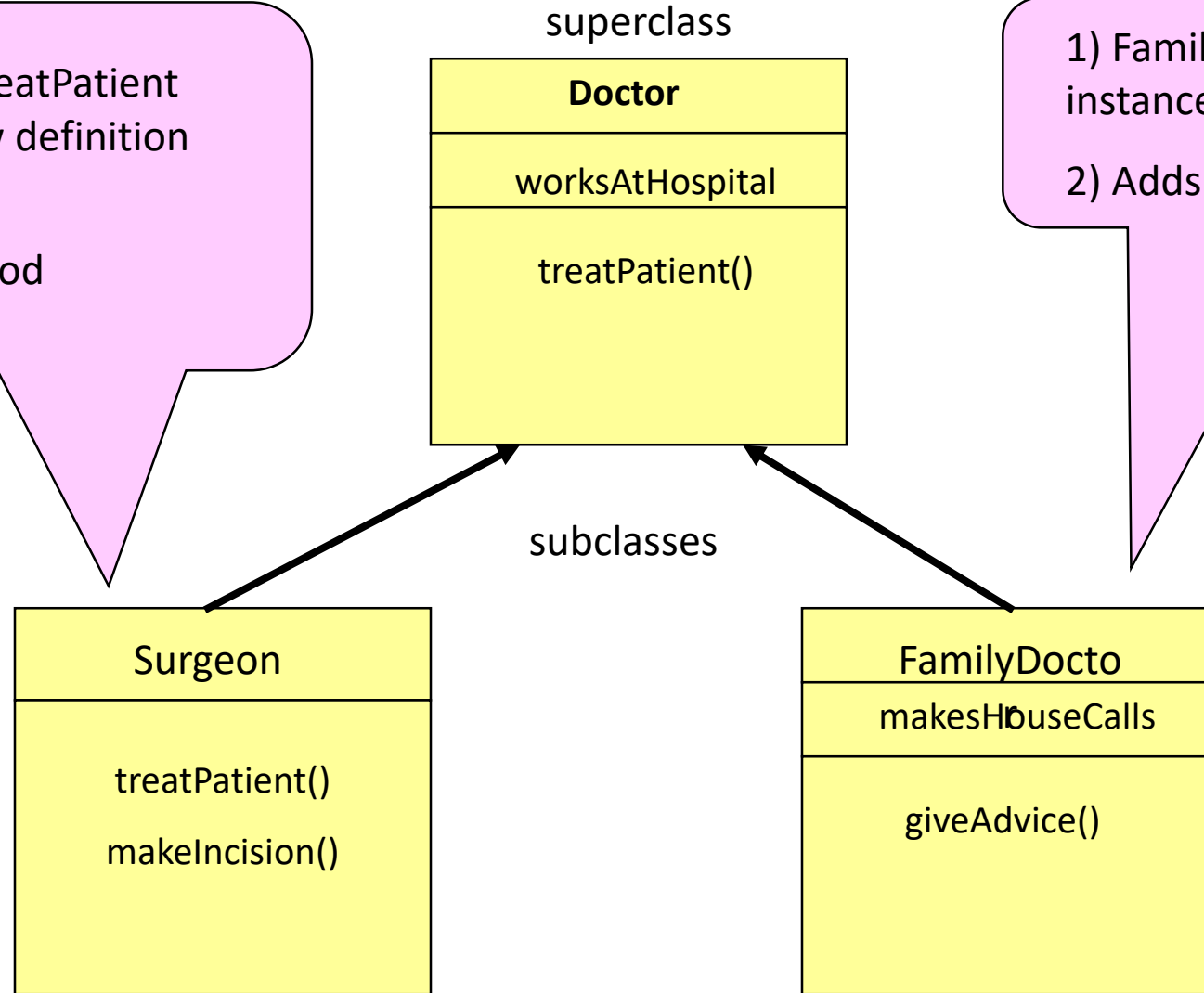
# METHOD OVERRIDING

1) Surgeon overrides treatPatient method i.e gives a new definition to the method

2) Adds one new method

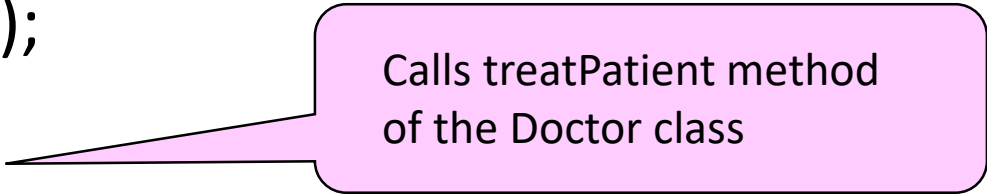
1) FamilyDoctor adds one new instance variable

2) Adds one new method




# METHOD OVERRIDING

- Doctor doctorObj = new Doctor();  
doctorObj.treatPatient()



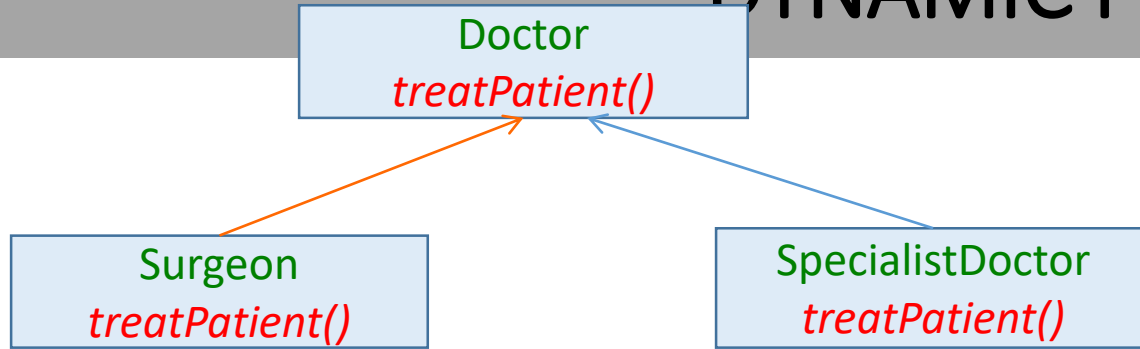
Calls treatPatient method  
of the Doctor class

- Surgeon surgeonObj = new Surgeon();  
surgeonObj.treatPatient()



Calls treatPatient method of  
the Surgeon class

# DYNAMIC POLYMORPHISM



Can we write this?

***Doctor obj = new Surgeon();***

```
obj = new SpecialistDoctor();  
obj.treatPatient();
```

**Yes.** A reference to a super class can refer to a sub class object

Now when we say *obj.treatPatient()*, which version of the method is called?

It calls **Surgeon's version of the treatPatient() method** as the reference is pointing to a Surgeon object. This is called as **Dynamic Polymorphism**.

If a super class reference is used to call a method, the method to be invoked is decided by the JVM at run-time, depending on the object the reference is pointing to.

As we have seen in the above example, even though *obj* is a reference to Doctor, it calls the method of Surgeon, as it points to a Surgeon object.

This is decided during run-time and hence termed as *dynamic or run-time polymorphism*.

# METHOD OVERRIDING

Invokes treatPatient()  
method of Surgeon  
class. Dynamic  
polymorphism

superclass

Doctor
worksAtHospital
treatPatient()

```
public void treatPatient(){  
...  
}
```

Valid because treatPatient()  
method of Doctor class is  
inherited into FamilyDoctor  
class

```
Doctor obj = new Sugeon();  
obj.treatPatient();  
obj.makeIncision();
```

```
Doctor obj = new FamilyDoctor();  
obj.treatPatient();  
new FamilyDoctor().treatPatient();  
obj.giveAdvise();
```

subclasses

Surgeon
treatPatient() makeIncision()

FamilyDoctor
makesHouseCalls giveAdvice()

```
public void treatPatient(){  
...  
}
```

**New methods of sub class cannot be invoked through  
super class reference**

# Accessing Superclass Members

If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword **super**.

If the *treatPatient()* method in the Surgeon class wants to do the functionality defined in Doctor class and then perform its own specific functionality, then

The *treatPatient ()* method in the Surgeon class should be written as:

```
public void treatPatient(){  
    super.treatPatient();  
  
    //add code specific to Surgeon  
}
```

This calls the super class version of *treatPatient()* and then comes back to do the sub-class specific stuff

To invoke super class method() that is overridden in the subclass :

***super.method();***

-Can be placed anywhere in the method unlike *super()* that has to be the first statement in a sub class constructor.

# Accessing the hidden field of super class

To access the hidden field of a super class

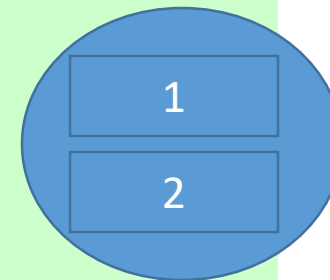
***super.instanceField***

```
class A {  
    int i = 1;  
    int f() {  
        return i;  
    }  
}  
class B extends A {  
    int i;          // This field shadows i in A  
    @Override  
    int f() {  
        i = super.i + 1;  // this.i = super.i + 1;  
        return super.f() + i;  
    }  
}
```

new B().f();

new A().f();

A a=new B();  
a.f();



Remember, this situation arises because , same field names exist in the super class as well as in the subclass which is not recommended.

# super keyword

*To summarize super keyword has 3 usages:*

1. `super()` : To invoke super class constructor from within sub class constructor
2. `super.method()` : To invoke super class method from within the overridden method of sub class
3. `super.field` : To access the super class instance field in the subclass method(data hiding).



Thank You!