GUI
Chapter 1-Modularity and object

Abed Safadi,Ph.D

Plan

- Themes
- Modularity
- Object

Themes

Object oriented programming

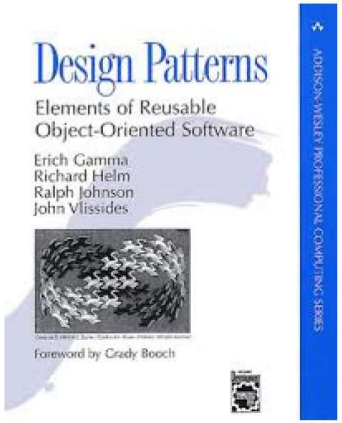
- motivation of the approach
- objects and classes
- inheritance, polymorphism and dynamic binding
- modeling principles

Bertrand Meyer :

Object-Oriented Software Construction, Prentice Hall.

Themes

- Design patterns
  - Creational
  - Structural
  - Behavioral



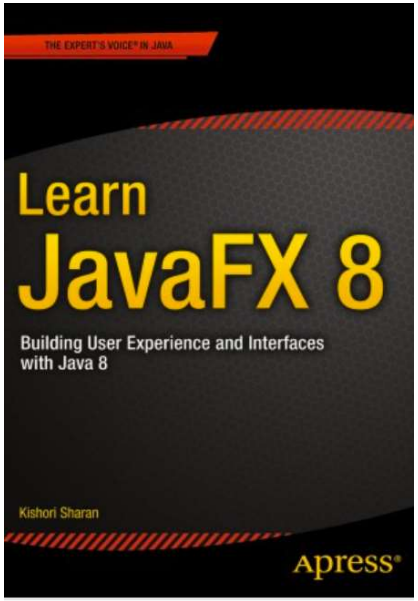
Erich Gamma, Richard Helm, Ralph Johnson & John Vlissid

Design Patterns : Elements of Reusable Object-Oriented Software.

.

Themes

- Graphical user interface
  - JavaFX
  - JavaFX Scene Builder



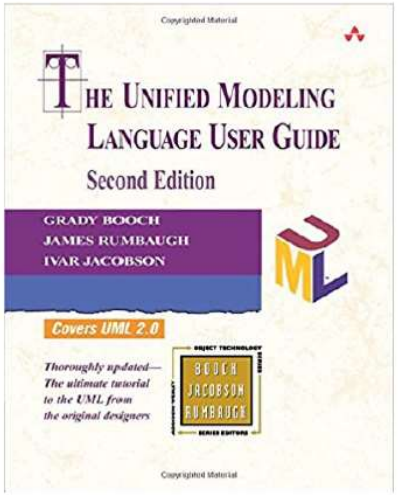
Kishori Sharan

Learn JavaFX 8
Building User Experience and Interfaces with Java 8

Other references

Robert C. Martin :

Clean Architecture : A Craftsman's Guide to Software Structure and Design



James Rumbaugh, Ivar Jacobson & Grady Booch :
The Unified Modeling Language Reference Manual, 2nd ed.
Dispo aussi en français :
UML 2.0.

Positioning of the GUI

Complementary with the software engineering course

Sharing points: software quality, design process

Concrete aspects seen in GUI : modeling, programming

mini-project

Implementation in java

One of the most requested languages

Sufficient to cover important GUI needs

Plan

- Themes
- Modularity
- Object

Modularity

- Modularity is a core concept in basically all forms of enginnering, including computer systems, both hardware and software
- Definition : « the degree to which a system's components may be separated and recombined»
- Let's walk through some examples

Examples of Modularity

- Computer Hardware
- Java Code
- Networking
- …..

Modularity in Programming and design

- Breaking down a complex system in to smaller parts of components
- Take a large problem and start splitting into smaller problems
  - Then iterate until the problems are « manageable»

Some strategies (high- and low level):

- Three-tier architecture→distinct major components(Model, view, controller)
- Object Oriented decompoistion→classes/models
- ….

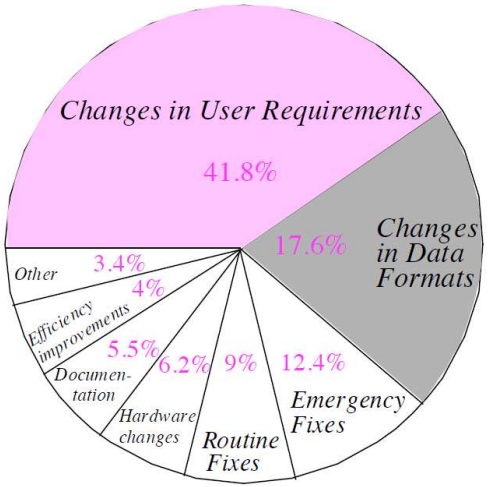
Goals of Modularity

- Extendable, Readable, reliable, reusable, maintainable components
- Smaller problems are easier to understand
- Easier to test/ debug
- Easier to update in future releases
- Distribution of effort team

Why is it so important?

- Maintenance is what happens after a software product has been delivered. But it is widely estimated that 70% of the cost of software is devoted to maintenance
  - 67% between 1976 and 1981
  - 75% between 1992 and 1998

[S.R. Sachs, Object-Oriented Software Engineering]



Breakdown of maintenance costs.

What can be considered a module?

- in the case of software, a module can be **any basic unit of code**

- This means different things in different languages:
  - C or Javascript-probably a function
  - Java or C#- probably a class

- if you can take a piece of code an encapsulate it in some way, then it can become a module

What makes a good module?

- key questions designes ask:
  - What should be a separate module or subprogram and why?
  - How do modules share information?
  - What should modules « know» about other modules or the rest of the system?
  - how is control organized in the system?

- These are very hard questions to answer!

- And software engineers have been working on them for a long time!

Ideal Qualities of a good Module

- Encapsulates just one system task or function
- Resuable
- Testable
  - Some say a module is the smallest chunk of code that can be tested independently
- Hides a design decision from the rest of the system
- Can be modified internally without affecting the rest of the system

Criteria of modulaire architecture

- **Decomposability**. : if it helps in the task of decomposing a software problem into a small number of less complex subproblems
- **Composability**. : if it favors the production of software elements which may then be freely combined with each other to produce new systems
- **Understandability** : if it helps produce software in which a human reader can understand each module without having to know the others,
- **Continuity**: a small change in a problem specification will trigger a change of just one module, or a small number of modules.
- **Protection**. : if it yields architectures in which the effect of an abnormal condition occurring at run time in a module will remain confined to that module,

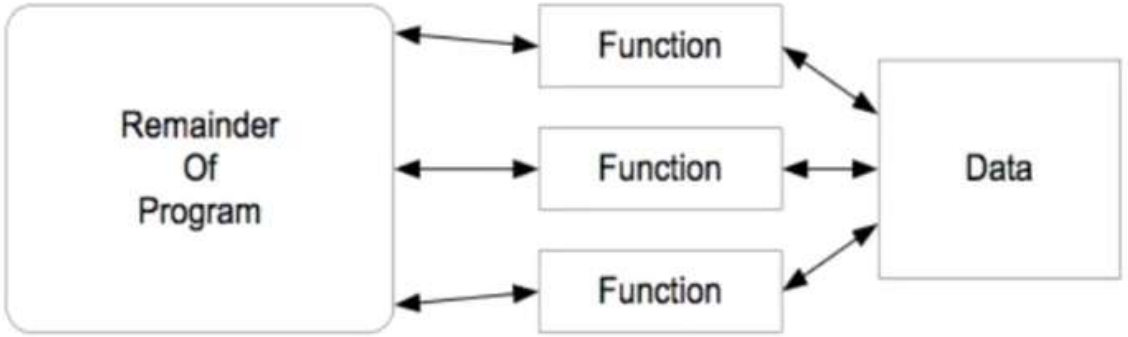
Criteria of modulaire architecture

Information Hiding: The primary goal is to prevent extensive modification to clients whenever the implementation details of a module or program are changed

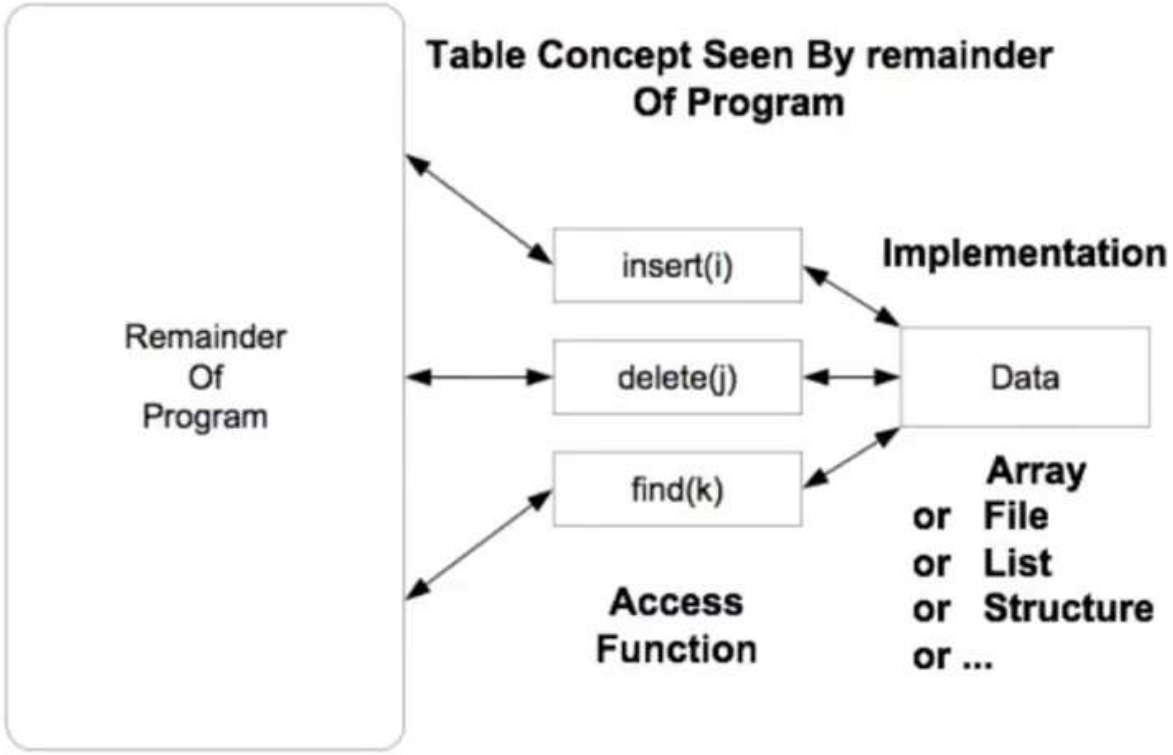
Information Hiding: is the principle of séparation of the design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed

Information Hiding

- Goal is to hide **design decision**
- Concept is extremly important to good software design
- Led to the concept of the modern object



Information Hiding



Criteria of modulaire architecture

- **Uniform Access:** the Uniform Access principle is in fact a design rule which influences many aspects of object-oriented design and the supporting notation (getter Setter methods).

- **The Open-Closed principle:**
  - A module is said to be open if it is still available for extension closed to modifications.
    - For example, it should be possible to expand its set of operations

Plan

- Themes
- Modularity
- Object

First contributions of the object-oriented

Concept of class which combines

- a data type
- an implementation

Class is the fundamental element that allows you to build

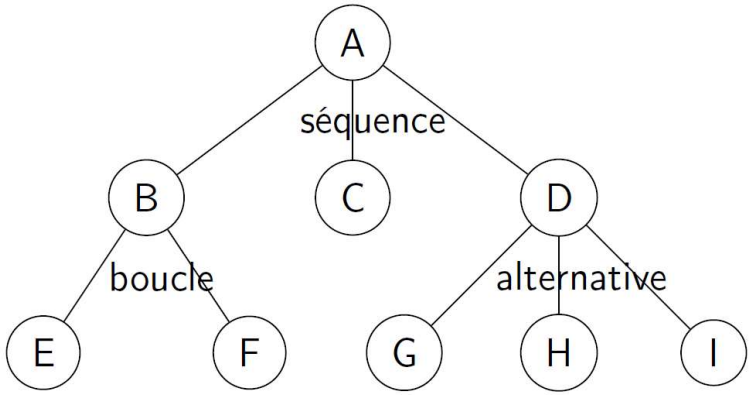
- software modules
- a sufficient set of usable types in the language

And gives us the means to respect the modularity criteria

Designing software: functional decomposition

Top-down, approach, classic in engineering

- Proceed by decomposition of the main function of the software into sub-problems



- Repeat the decomposition until you get basic sub problems that are easy to solving it

Problems of functional decomposition

Difficult to derive a modular architecture

- the software can have multiple main functions, how to choose one?
- in a functional view, with a time division (B then C….), certain distinct parts are difficult to separate from each other, eg user interface / functional core
- often leads to a specific decomposition, the elements of which are not very reusable

Object-oriented Software Construction

It is a **bottom-up** process of building robust and extendible solutions to parts of the problem, and combining them into more and more powerful assemblies — until the final assembly which yields a solution of the original problem

Contribution:

- **Extendibility**, **Reusability** (a set of functions "associated with a type" is more reusable than an isolated function)

ISSUES

- How to find the relevant object types.
- How to describe the relations between object types.

……

Back to the concept of class

Pattern used to create objects, a class combines

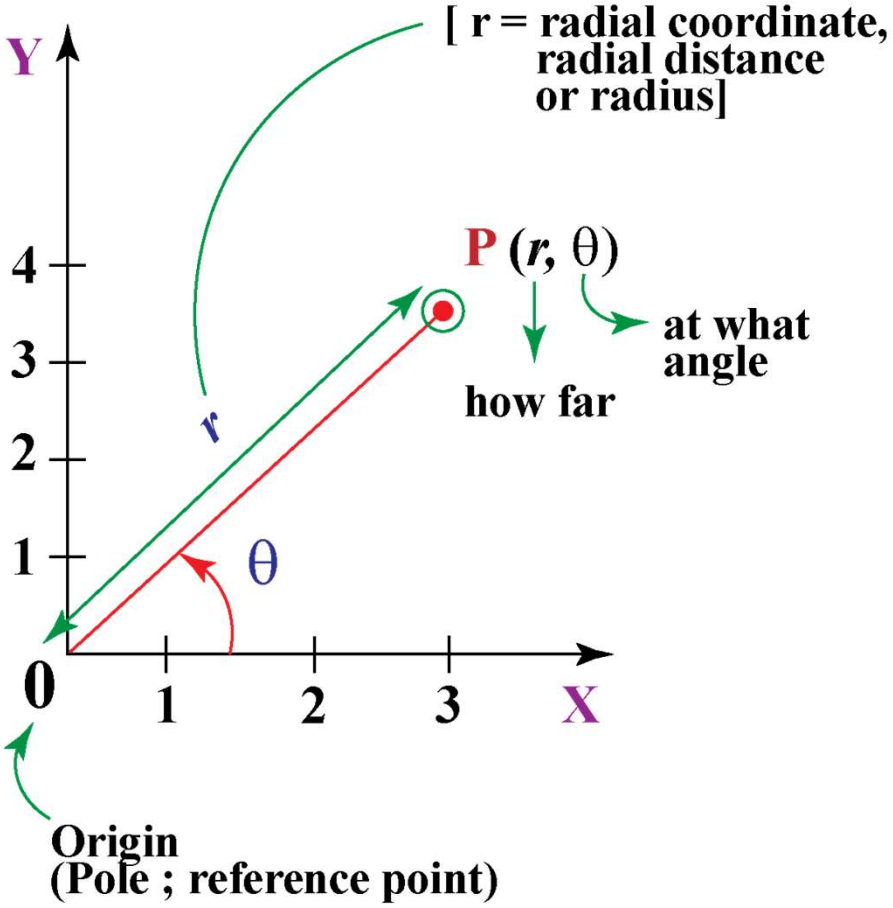
- **a structured data type**
  - It consists of fields called attributes
  - whose attributes each have an existing type
- **a set of operations applicable to this type**
  - these operations are called methods
  - various methods to build, consult and order

The class is the source code written by the programmer

A class does not exist in memory at runtime

- it is used as a type to create objects (instances of the class)
- these instances exist in memory

Coordinate system example

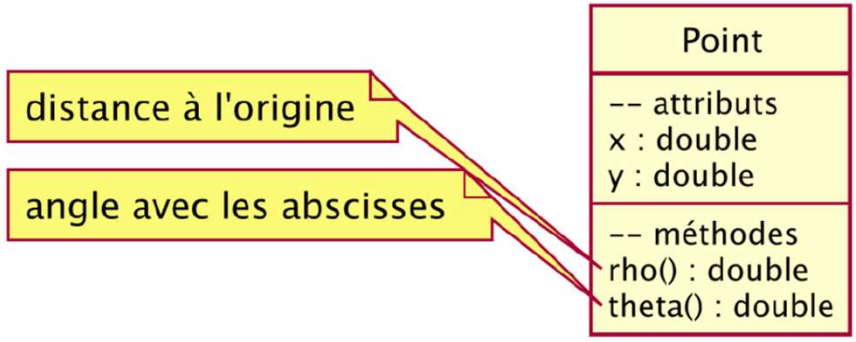


Class example

En java

```
class Point {
  double x, y;

  double rho() {
    return Math.sqrt(x*x+y*y);
  }

  double theta() {
    return Math.atan2(y, x);
  }
}
```

En UML



Object concept

An object is an instance of a class

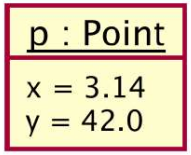
- created by an operator or a special function of the language
- occupies a space in memory
  - meta information (type of object)
  - attribute values
  - link table (s) (methods)
- Declared and manipulated
  - by reference (typically for sharing)

Object example

En java

```
// La classe Point est associée
// à un type de références
Point p;

// Opérateur new : création
// renvoie une référence
p = new Point();

// Accès aux attributs avec .
p.x = 3.14;
p.y = 42.0;
```

En UML



Methods

A method is associated with an object

- can only be called from this one
- this object is an implicit parameter of the call
  - always exists (otherwise, by definition, the call is impossible)
  - is referred to by the reference “this” in java

Example of a method call

```
Point p;

p = new Point();
p.x = 3.14;
p.y = 42.0;

System.out.println("Rho : " + p.rho());
System.out.println("Théta : " + p.theta());
```

Constructor

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created.

In java, this task is completed by a constructor:

- method without return type, with the same name as the class
- can be overloaded
- called implicitly by the new or explicitly by this
- implicit empty constructor if the class has none

Constructor Example

```
class Point {
  double x, y;

  Point() {
    // Appel de l'autre
    // constructeur
    this(0,0);
  }

  Point(double x,
        double y) {
    this.x = x;
    this.y = y;
  }
}
```

```
Point p;

p = new Point(3.14, 42.0);
System.out.println(
  "Rho : " + p.rho() + "\n"
  + "Théta : " + p.theta()
  );

p = new Point();
System.out.println(
  "Rho : " + p.rho() + "\n"
  + "Théta : " + p.theta()
  );
```

Review of the chosen example

Our Point class example goes against the rules and principles of modular architecture

No uniform access

- access to attributes via p.x and p.y
- access to methods via p.rho () and p.theta ()

No information hiding

- attributes and methods accessible by the main program

Uniform access, why is it important

Storing Cartesian coordinates is only an implementation choice

```
class Point {
  double x, y;

  double rho() {
    return
      Math.sqrt(x*x+y*y);
  }

  double theta() {
    return
      Math.atan2(y, x);
  }
}
```

```
class Point {
  double rho, theta;

  double x() {
    return
      rho*Math.cos(theta);
  }

  double y() {
    return
      rho*Math.sin(theta);
  }
}
```

No direct access to attributes

- we thus hide the implementation choices

```
class Point {
  double rho, theta;

  double x() {return rho*Math.cos(theta);}
  double y() {return rho*Math.sin(theta);}
  void setX(double x) {double y = y();
    rho = Math.sqrt(x*x+y*y); theta = Math.atan2(y, x);
  }
  void setY(double y) {double x = x();
    rho = Math.sqrt(x*x+y*y); theta = Math.atan2(y, x);
  }

  double rho() {return rho;}
  double theta() {return theta;}
  void setRho(double r) {rho = r;}
  void setTheta(double t) {theta = t;}
}
```

Hidden implementation

On the side of the main program we only go through methods

- the choice of attributes is hidden
- the developer is free to change the implementation

```
Point p;

p = new Point();
p.setX(3.14);
p.setY(42.0);

System.out.println("Rho : " + p.rho());
System.out.println("Théta : " + p.theta());
```

Information Hiding

Access level modifiers determine whether other classes can use a particular field or invoke a particular method.

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
(default) *no modifier*	Y	Y	N	N
`private`	Y	N	N	N

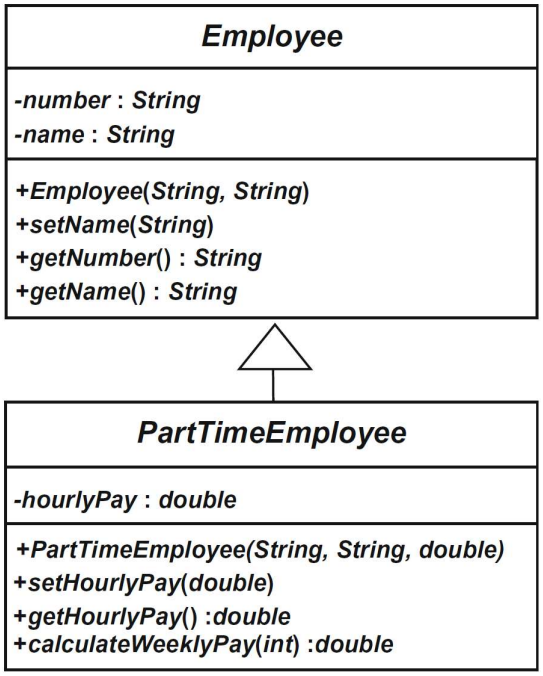
Example

```
class Point {
  private double rho;
  private double theta;
  double x() {return rho*Math.cos(theta);}
  double y() {return rho*Math.sin(theta);}
  void setX(double x) {double y = y();
    rho = Math.sqrt(x*x+y*y); theta = Math.atan2(y, x);
  }
  void setY(double y) {double x = x();
    rho = Math.sqrt(x*x+y*y); theta = Math.atan2(y, x);
  }

  double rho() {return rho;}
  double theta() {return theta;}
  void setRho(double r) {rho = r;}
  void setTheta(double t) {theta = t;}
}
```

Inheritance

Inheritance is the sharing of attributes and methods among classes. We take a class, and then define other classes based on the first one. The new classes inherit all the attributes and methods of the first one, but also have attributes and methods of their own

An inheritance relationship is a hierarchical relationship. The class at the top of the hierarchy—in this case the **Employee** class **is** referred to as the **superclass** (or base class) and the **PartTimeEmployee as the subclass**



Java Code

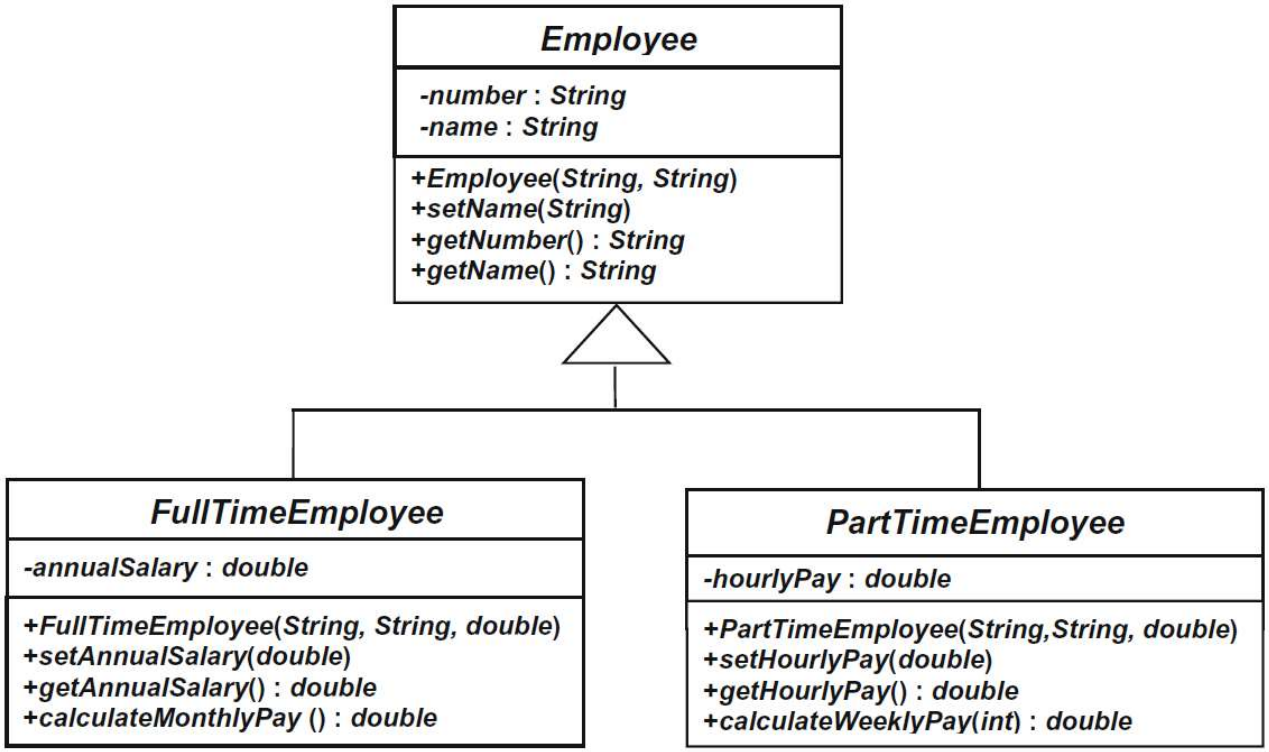
Employee

```
public class Employee
{
    private String number;
    private String name;
    public Employee(String numberIn, String nameIn)
    {
        number = numberIn;
        name = nameIn;
    }

    public void setName(String nameIn)
    {
        name = nameIn;
    }

    public String getNumber()
    {
        return number;
    }

    public String getName()
    {
        return name;
    }
}
```

PartTimeEmployee

```
public class PartTimeEmployee extends Employee // this class is a subclass of Employee
{
    private double hourlyPay; // this attribute is unique to the subclass

    // the constructor
    public PartTimeEmployee(String numberIn, String nameIn, double hourlyPayIn)
    {
        super(numberIn, nameIn); // call the constructor of the superclass
        hourlyPay = hourlyPayIn;
    }

    // these methods are also unique to the subclass
    public double getHourlyPay()
    {
        return hourlyPay;
    }

    public void setHourlyPay(double hourlyPayIn)
    {
        hourlyPay = hourlyPayIn;
    }

    public double calculateWeeklyPay(int noOfHoursIn)
    {
        return noOfHoursIn * hourlyPay;
    }
}
```

Inheritance – extended Version



Inheritance – extended Version

FullTimeEmployee

```
public class FullTimeEmployee extends Employee
{
    private double annualSalary;

    public FullTimeEmployee(String numberIn, String nameIn, double salaryIn)
    {
        super(numberIn,nameIn);
        annualSalary = salaryIn;
    }

    public void setAnnualSalary(double salaryIn)
    {
        annualSalary = salaryIn;
    }

    public double getAnnualSalary()
    {
        return annualSalary;
    }

    public double calculateMonthlyPay()
    {
        return annualSalary/12;
    }
}
```

Main Program

EmployeeTester

```
public class EmployeeTester
{
    public static void main(String[] args)
    {
        FullTimeEmployee fte = new FullTimeEmployee("A123", "Ms Full-Time", 25000);
        PartTimeEmployee pte = new PartTimeEmployee("B456", "Mr Part-Time",30);
                Employee ffte = new FullTimeEmployee("A123", "Ms Full-Time", 25000);
                Employee ppte = new PartTimeEmployee("B456", "Mr Part-Time",30);
    }
}
```

EmployeeTester

```
public class EmployeeTester
{
    public static void main(String[] args)
    {
        FullTimeEmployee fte = new FullTimeEmployee("A123", "Ms Full-Time", 25000);
        PartTimeEmployee pte = new PartTimeEmployee("B456", "Mr Part-Time",30);
        testMethod(fte); // call testMethod with a full-time employee object
        testMethod(pte); // call testMethod with a part-time employee object
    }

    static void testMethod(Employee employeeIn) // the method expects to receive an Employee object
    {
        System.out.println(employeeIn.getName());
    }
}
```

Type Polymorphism

- **Polymorphism** is not required in order to achieve **inheritance**
  - It means "many forms“ and occurs when we have many classes that are related to each other by **inheritance**

- We saw in the *EmployeeTester* how it was possible to call this method and send in objects of any subclass of Employee such as Full- TimeEmployee or PartTimeEmployee. Because objects of a subclass are of the same type as the superclass, a method can effectively receive parameters of more than one type
  ➔ this can be regarded as a kind of polymorphism—this is known as ***subtype polymorphism***.

Polymorphism

If class B inherits from class A

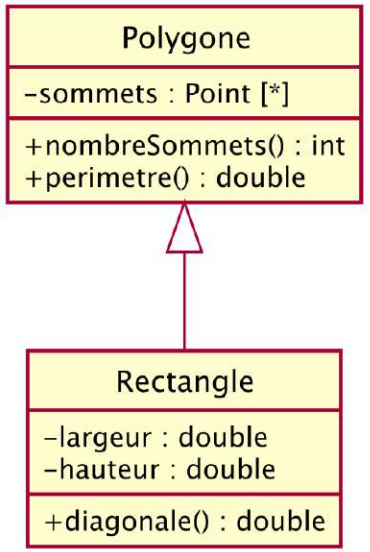
- Class B « IS-A" Class A
- any method m of A can be invoked on an instance of class B

Polymorphism consists in exploiting this by providing a B in the expressions "which expect" an A

```
A a=new B();
```

Example 2

```
class Polygone {
  private Point [] sommets;
  ...
  public int nombreSommets() { ...
  public double perimetre() { ...
}

class Rectangle extends Polygone {
  private double largeur;
  private double hauteur;
  ...
  public double diagonale() { ...
}
```



Java code

```
Point calculeBarycentre(Polygone p) {
   ...
}
...
Polygone p;
Rectangle r = new Rectangle(...);

// affectation polymorphique, r est vu comme un polygone
p = r;
// ok, Polygone dispose de la méthode perimetre
System.out.println("Périmètre : " + p.perimetre());
// En revanche, à travers p, l'objet référencé
// semble être un Polygone, impossible d'appeler
// p.diagonale()
// même si le Rectangle caché derrière dispose de
// cette méthode

// paramètre polymorphique
Point o = calculeBarycentre(r);
```

Inheritance and modularity

The ingredients of the Inheritance

- new definitions and redefinitions (Overriding)
- polymorphism
- dynamic binding

Allow the principle of opening / closing to be applied:

- the parent class (SuperClass) is not changed by inheritance, as a module it remains closed to modifications
- the child class (SubClass) inherits the characteristics of its parents but can change and complete them, it constitutes the same module but open to extensions

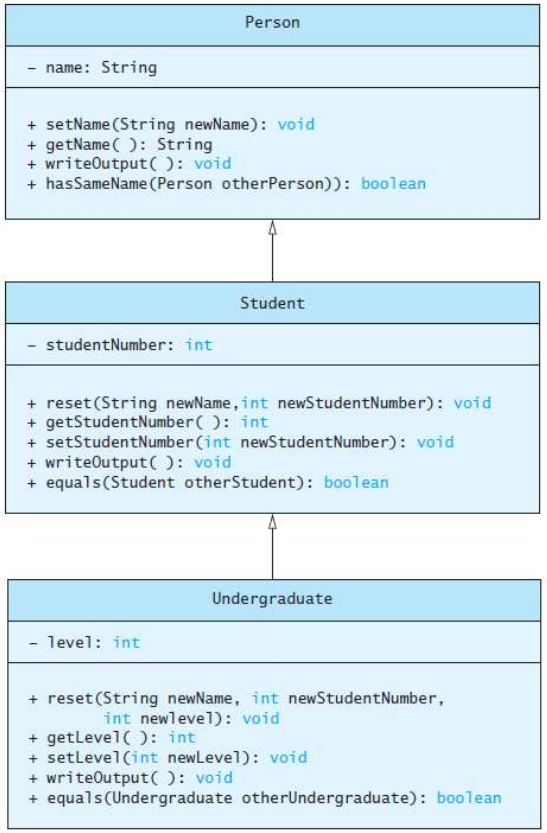
Polymorphic variables – Example

Consider an array of **Person**

```
Person[] people = new Person[4];
```

Since **Student** and **Undergraduate** are types of **Person**, we can assign them to **Person** variables

```
people[0] = new Person("Libby", "Jeremy");
people[1] = new Student("DeBanque, Robin", 8812);
people[2] = new Undergraduate("Cotty, Manny", 8813, 1);
Student s = new Student("Bob, Pazo", 8814);
people[3] = s;
```

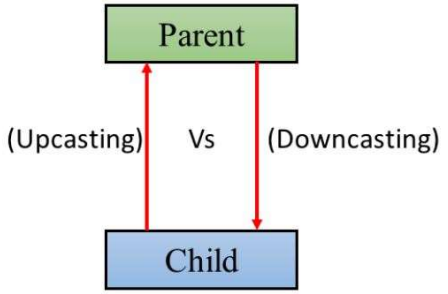


Java Casting

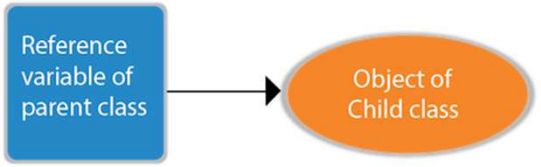
- Type casting is a way of converting data from one data type to another data type
- In Java, we can cast both reference and primitive data types

Upcasting

- **Parent p = new Child():**
- **Upcasting: Casting (converting) a child object to a parent object**



- **Upcasting**: When the **reference** variable of **Parent** class refers to the **object** of **Child** class
- Assigning an object of a derived class to a variable of a base class is called **upcasting** because it is like a type cast to the type of the base class



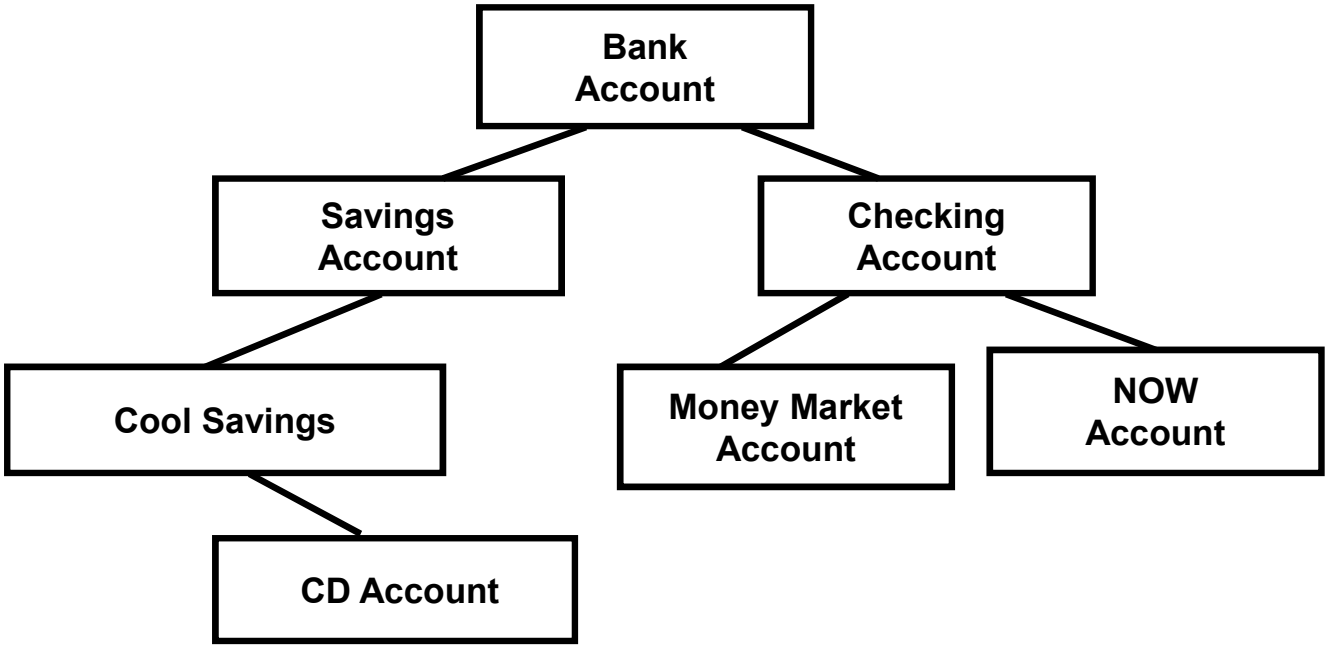
Upcasting Syntax

- Casting an object from a subclass to a superclass

```
4   Animal animal = new Animal();
5   Cat cat = new Cat();
6
7   animal = cat;                   //Implicit way
8   animal = (Animal) cat;          //Explicit way
9
10  //OR
11
12  Animal animal = new Cat();
```

Polymorphic variables – Example 3

The Banking Class Hierarchy



Polymorphic variables

Imagine a bank needs to manage all of the accounts Rather than maintain seven separate arrays, one each for: `Bank_Accounts`, `Savings_Accounts`, `Cool_Savings`, `CD_Accounts`, `Checking_Accounts`, `NOW_accounts`, and `Money_Market_Accounts`

We can maintain only one array of Bank Accounts

Overriding

- **Overriding**: Replacing an inherited method with another having **the same signature**
- It allows you to make changes in the method definition for the derived classes
  - Use a specific method to perform different tasks: perform a single action in different ways
  - Ability for the same code to be used with different types of objects and behave differently with each

Overriding Prerequisites

- **Certain things to remember in method overriding are:**
  - The method must have the same name as one mentioned in the parent class
  - The method must have the same parameter as one mentioned in the parent class
  - There must be the inheritance: the **IS-A relationship**

Overriding Example

```
3  public class Animal{
4
5      public void makingSound(){
6          System.out.println("This animal making a sound.");
7      }
8  }
9
10 public class Cat extends Animal{
11
12     public void makingSound(){
13         System.out.println("Meow meow");
14     }
15 }
```

```
Cat tom = new Cat();
tom.makingSound();    //output: Meow meow
```

The main advantage of Overriding is that the class can give its own specific implementation to an inherited method and without modifying the parent class code

Overriding Example

Given:

```
Person[] people = new Person[4];
people[0] = new Person(“Libby”, “Jeremy");
people[1] = new Student("DeBanque, Robin", 8812);
people[2] = new Undergraduate("Cotty, Manny", 8813, 1);
```

When invoking:

```
people[1].writeOutput();
```

Which `writeOutput()` is invoked, the one defined for `Student` or the one defined for `Person`?

Answer: the one defined for `Student`

Dynamic binding

- In an inheritance relation and in the presence of polymorphism, the type of binding (Link) used affects the call of methods

- The sub-classs can redefine (overriding) a method *m* of the parent class (superclass) (different or more efficient behavior)
- a reference *r* of the Super class, can designate an object of the SubClass and *r* can be used to call *m*. So if the binding is
  - dynamic, this is the version of the sub class that is called
  - static, it is that of the Super class

- Usually dynamic binding is desired
  - default behavior in java
  - binding is static for methods declared final

Dynamic binding example

```
class Polygone {
  Point [] sommets;
  ...
  double perimetre() {
    // parcours des cotés ...
}

class Rectangle extends
        Polygone {
  double largeur;
  double hauteur;
  ...
  @Override // redéfinition
  double perimetre() {
    return 2*largeur +
      2*hauteur;
  }
}
```

```
Polygone p;
Rectangle r;

p = new Polygone(...);
r = new Rectangle(...);
double perim;

// méthode de Polygone
perim = p.perimetre();

p = r;
// méthode de Rectangle
perim = p.perimetre();
```

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }
   public static void m(Object x) {
    System.out.println(x.toString());
  }
}
class GraduateStudent extends Student {
}
class Student extends Person {
  public String toString() {
    return "Student";
  }
}
class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

An object of a subtype can be used wherever its supertype value is required

Method *m* takes a parameter of the Object type. You can invoke it with any object

Which implementation of the *toString()* method is used will be determined dynamically at run-time - *dynamic binding*

- **Overloading**: Two or more methods with **different signatures**
  - To achieve overloading, **Static binding/Compile-Time binding/Early binding** must be used
  - On compile time, the compiler checks the types of the parameters passed to a method call to decide which method having the given name should be invoked
    - Hence, the **static binding**

Static Binding

- When the decision of which definition of a method to use is made at compile time, that is called **static binding -** When the type of object is determined at compile-time

- `private`, `final`, and `static` methods use static binding
  - Static binding uses the class type information for binding
  - Dynamic binding uses object type for binding

Overloading

Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters or both)

```
class Point {
  double x, y;

  void reset() {
    reset(0,0);
  }

  void reset(double x,
             double y) {
    this.x = x;
    this.y = y;
  }
}
```

```
Point p = new Point();

p.reset(3.14, 42.0);
System.out.println(
  "Rho : " + p.rho() + "\n"
  + "Théta : " + p.theta()
  );

p.reset();
System.out.println(
  "Rho : " + p.rho() + "\n"
  + "Théta : " + p.theta()
  );
```

Overloading a Method

- **Overloading** is when two/more methods have the same method name in the same class
  - Overriding of functions occurs when one class is inherited from another class
  - **Overloading can occur without inheritance**
- To be valid, any two definitions of the method name must have different *signatures*
  - Differing signatures must have **different numbers and/or types of parameters**

```
setDate(int, int, int)
setDate(String, int, int)
setDate(int)
```

You can't overload based on the type returned

- Java does not permit methods with the same name and different return types in the same class

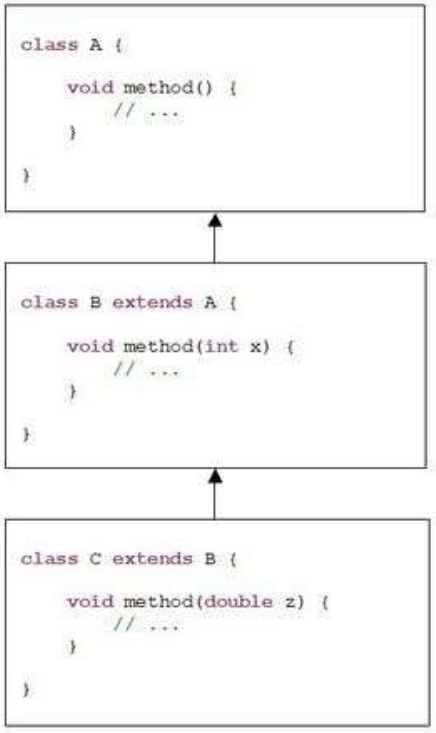
Overriding vs. Overloading

```
3  class Animal {
4    public void animalSound() {
5      System.out.println("The animal makes a sound");
6    }
7  }
8
9  class Pig extends Animal {
10   public void animalSound() {
11     System.out.println("The pig says: wee wee");
12   }
13 }
14
15 class Dog extends Animal {
16   public void animalSound() {
17     System.out.println("The dog says: bow wow");
18   }
19 }
```

```
3  class Test {
4  public static void main(String args[]) {
5        myPrint(5);
6        myPrint(5.0);
7        }
8
9        static void myPrint(int i) {
10           System.out.println("int i = " + i);
11           }
12
13       // same name, different parameters
14       static void myPrint(double d) {
15       System.out.println("double d = " + d);
16       }
17 }
```

Overloading a Method

- There are the following differences between overriding and overloading a method:
- When **overridden**, superclass and subclass methods have **the same name and the same signature: same number and types of parameters as in the base class**
- When **overloaded**, the methods of the superclass and subclass have **the same name but different signatures**
- **When the derived class overloads the original method, it still inherits the original method from the base class as well**

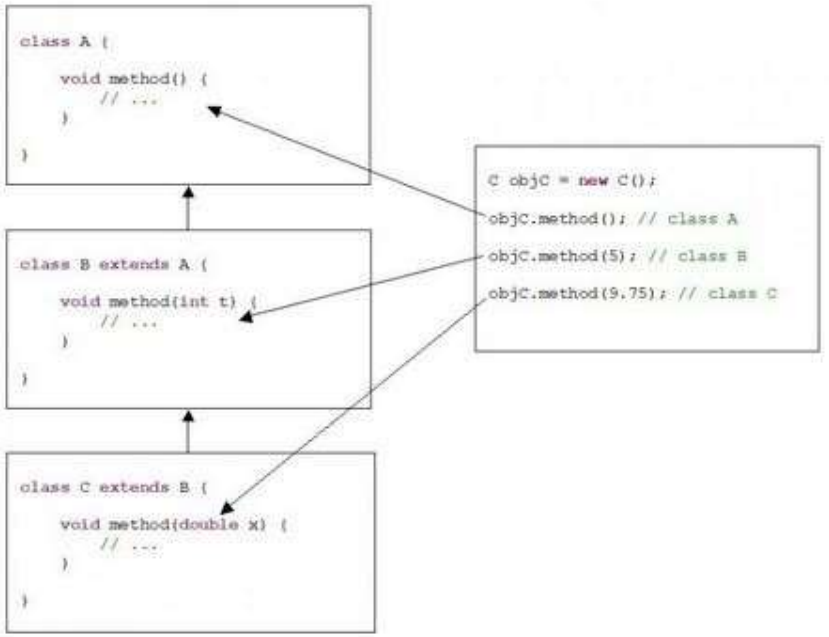


Overloading a Method

A method is considered to be **overloaded** (not overrided) if:

- In different classes there is a method with the same name
- The signature of the method parameters in each class is different

Thus, methods with different signatures are considered overloaded and not overrided



```java
// superclass
class A {
    void method() {
        // method of class A
        System.out.println("Class A. Method without parameters.");
    }
}

// subclass of class A
class B extends A {
    void method(int t) {
        // method of class B
        System.out.println("Class B. Method with 1 parameter of type int: "+t);
    }
}

// subclass of class B
class C extends B {
    void method(double x) {
        // method of class C
        System.out.println("Class C. Method with 1 parameter of type double: "+x);
    }
}
```

```java
public class Test {

    public static void main(String[] args) {
        C obj = new C(); // instance of class C

        obj.method(); // calling a method without parameters of class A
        obj.method(5); // calling a method of class B
        obj.method(9.75); // calling a method of class C
    }
}
```

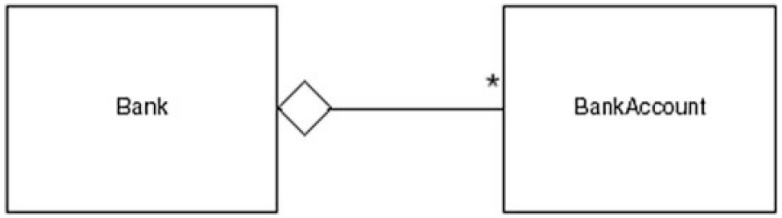
```
Class A. Method without parameters.
Class B. Method with 1 parameter of type int: 5
Class C. Method with 1 parameter of type double: 9.75
```

Aggregation/Composition

- When one object itself consists of other objects, this relationship is called aggregation. This association, represented in UML by a diamond, is often referred to as a part-of relationship. For example, the association between a car and the passengers in the car is aggregation.

- Composition (represented by a filled diamond) is a special, stronger, form of aggregation whereby the “whole” is actually dependent on the “part”. For example, the association between a car and its engine is one of composition, as a car cannot exist without an engine

Aggregation.



Bank
-list: ArrayList <BankAccount>
-search(String) : int *+getTotal() : int* *+getItem(String) : BankAccount* *+addAccount(String, String) : boolean* *+depositMoney(String, double) : boolean* *+withdrawMoney(String, double) : boolean* *+removeAccount(String) : boolean*

What does the following code display?

```
public abstract class A {
    abstract String un(int n);
    abstract String un(char x);
    abstract String un(String s);

    String deux() {
        return "A deux";
    }
}
```

```
public class B extends A {
    String un(int n) {
        return un(Integer.toString(n));
    }
    String un(char x) {
        return "B un " + x;
    }
    String un(String s) {
        return "B un " + s;
    }
}
```

```
public class C extends B {
    String un(String s) {
        return "C un " + s;
    }
}
```

```
public class ABC {
    public static void main(String [] args) {
        A objet = new C();
        System.out.println(objet.un(42));
        System.out.println(objet.un('X'));
        System.out.println(objet.un("42"));
        System.out.println(objet.deux());
    }
}
```

Resultt

```
public class ABC {
    public static void main(String [] args) {
        A objet = new C();
        System.out.println(objet.un(42));
        System.out.println(objet.un('X'));
        System.out.println(objet.un("42"));
        System.out.println(objet.deux());
    }
}
```

Output:

C un 42
B un X
C un 42
A deux

Questions?