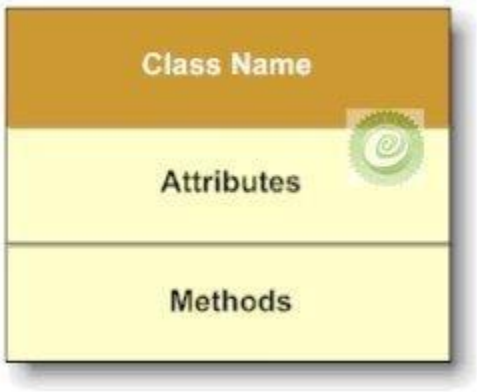
UML2 Class Diagram in Java

Posted on December 28, 2012 by IdioTechie | 15 Comments



In the modelling world Class diagram forms the major chunk of the Unified Modelling Language (UML) architecture diagram. In this article we are planning to show some of the key usage of the class diagram and how they can be represented in Java. This article can be used as a reference for your modelling.
UML was developed by Grady Booch, Ivar Jacobson, Jim Rumbaugh at Rational Software in 1990. To get more details on the UML development, it's progress and story of standardisation by *OMG* (please don't confuse this with the current SMS language of 'Oh My God', it is Object Management Group (*OMG*)) you can refer to Martin Fowler's UML Distilled book . He has explained very clearly about the practical use of the various UML diagrams in modelling.
In this article we will go straight to the point making it a crisp reference document.

Class Diagram Example:

Class Diagram Example

Car.java

?

Code:

```
1  public class Car {
2      private String carColor;
3      private double carPrice = 0.0;
4      public String getCarColor(String model) {
5          return carColor;
```

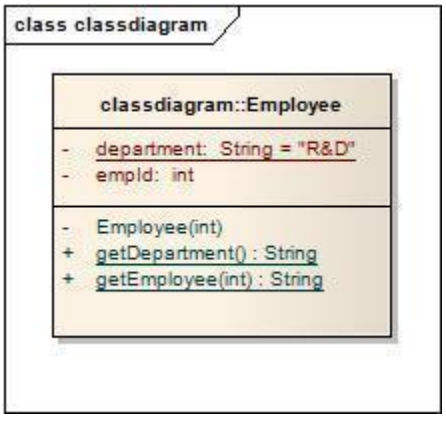
```
6       }
7
8       public double getCarPrice(String model) {
9           return carPrice;
10      }
11  }
```

The above example of Car class is self explanatory. The Car class has private instance variables carColor, carPrice denoted by (-) in the UML Class diagram. Similarly if this was public then it would have been represented as (+), if was protected then it is denoted by (#). The package visibility is defined by (~).

Java visibility	UML Notation
public	+
private	–
Protected	#
package	~

The return type of the instance variables or the methods are represented next to the colon (:) sign.
Structure:
[visibility] [attribute name] [multiplicity] [:type [=default value]] {property string}
Example: carPrice : double = 0.0

Representing Static variable or static operation:
The static data is represented with an underline. Let's take the below example.



Class diagram

?

Code:

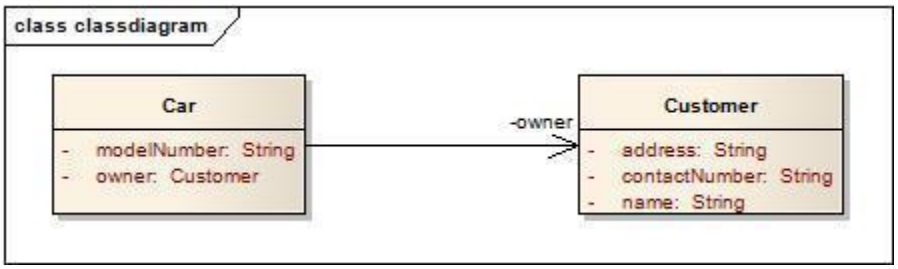
```
1  public class Employee {
2      private static String department = "R&D";
3      private int empId;
4      private Employee(int employeeId) {
5          this.empId = employeeId;
6      }
7      public static String getEmployee(int emplId) {
8          if (emplId == 1) {
9              return "idiotechie";
10         } else {
11             return "Employee not found";
12         }
13     }
14     public static String getDepartment() {
15         return department;
16     }
17 }
```

Association:
The association represents the static relationship between two classes along with the multiplicity. E.g. an employee can have one primary address associated with it but can have multiple mobile numbers.
Association are represented as thin line connecting two classes. Association can be unidirectional (shown by arrow at one end) or bidirectional (shown by arrow at both end).
Multiplicity defines how many instances can be associated at any given moment.

0..1	No instances or one instance	A flight seat can have no or one passenger only

1	Exactly one instance	An order can have only one customer
0..* or *	Zero or more instances	A class can have zero or more students.
1..*	One or more instances (at least one)	A flight can have one or more passenger

The unidirectional relationship shows that the source object can invoke methods of the destination class. In Java a possible example can be the instance variable of source class referencing the destination class.



Association Example

?

Code:

```
1  public class Customer {
2  private String name;
3  private String address;
4  private String contactNumber;
5  }
6
7  public class Car {
8      private String modelNumber;
9      private Customer owner;
10 }
```

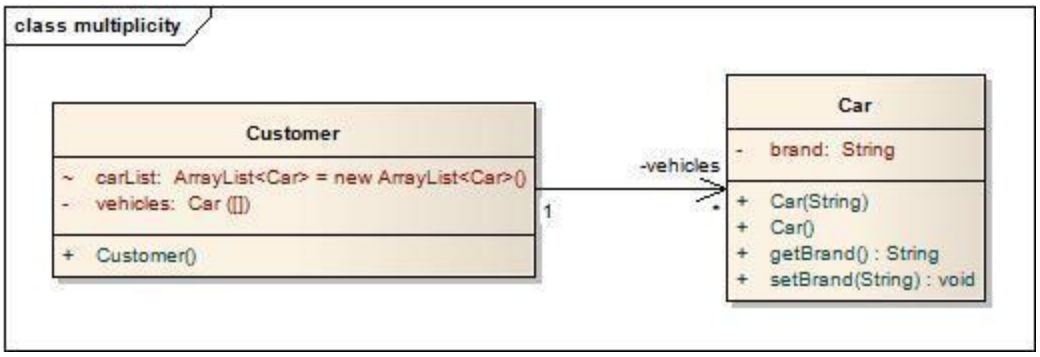
Let's look at an example of bidirectional association:

Bidirectional association
?

Code:

```
1  public class Customer {
2  private String name;
3  private String address;
4  private String contactNumber;
5  private Car car;
6  }
7
8  public class Car {
9      private String modelNumber;
10     private Customer owner;
11 }
```

In the bidirectional association each of the class in this relationship refers to each other by calling each others method. In the above Java example it is depicted as instance variable of Car class in called inside the Customer class and vice versa.
In the above example the car and owner refers to the roles and is depicted by the name of instance variable in the code.
Multiplicity:
Assume a scenario where a customer has multiple cars. How do we represent this situation in Java and UML?



Multiplicity in association

The above diagram explains a unidirectional association with a one to may relationship. Both use of ArrayList and Array is for illustration purposes only.

Car.java

?

Code:

```
1  public class Car {
2      private String brand;
3
4      public Car(String brands){
5          this.brand = brands;
6      }
7      public Car() {
8      }
9      public String getBrand() {
10         return brand;
11     }
12
13     public void setBrand(String brand) {
14         this.brand = brand;
```

```
15      }
16
17  }
```

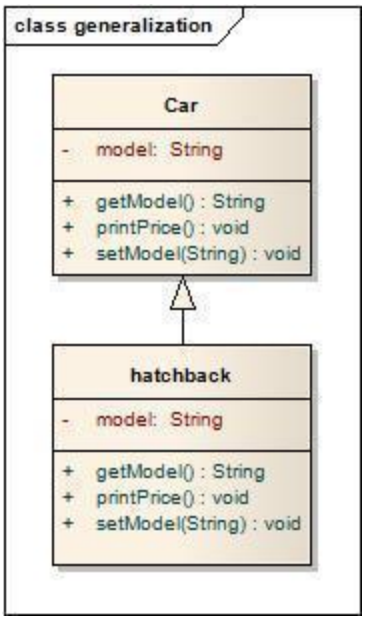
Customer.java

?

Code:

```
1   public class Customer {
2       private Car[] vehicles;
3       ArrayList<Car> carList = new ArrayList<Car>();
4       public Customer(){
5           vehicles = new Car[2];
6           vehicles[0] = new Car("Audi");
7           vehicles[1] = new Car("Mercedes");
8
9           carList.add(new Car("BMW"));
10          carList.add(new Car("Chevy"));
11      }
12  }
```

Generalization
This property represents the inheritance feature of the object oriented concept. In Java this can relate to the “extends” keyword. The inheritance should ideally follow the Liskov Substitution Principle i.e. the subtype should be able to substitute for its supertype. It helps to make the code implicitly follow the Open Close Principle i.e. Open for extension but closed for modification.



Generalization

?

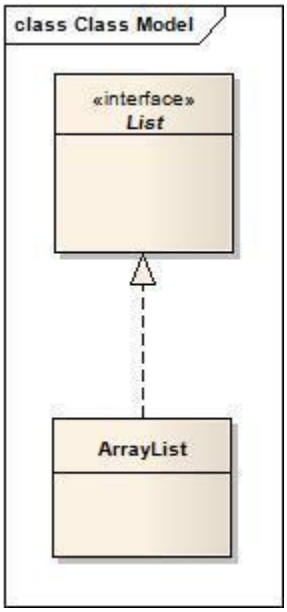
Code:

```
1   public class Car {
2       private String model;
3       public void printPrice() {
4       }
5       public String getModel() {
6           return model;
7       }
8       public void setModel(String model) {
9           this.model = model;
10      }
11  }
12
```

```
13  public class hatchback extends Car {
14      private String model;
15      public void printPrice() {
16          System.out.println("Hatchback Price");
17      }
18      public String getModel() {
19          return model;
20      }
21      public void setModel(String model) {
22          this.model = model;
23      }
24  }
```

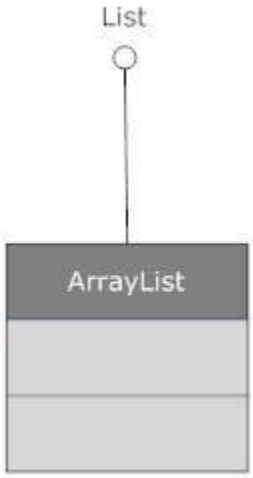
Realization:

This is related to the relationship between the class and the interface. The realization is equivalent to the "*implements*" keyword in Java.



Realization in Java

Realization can also be represented as :



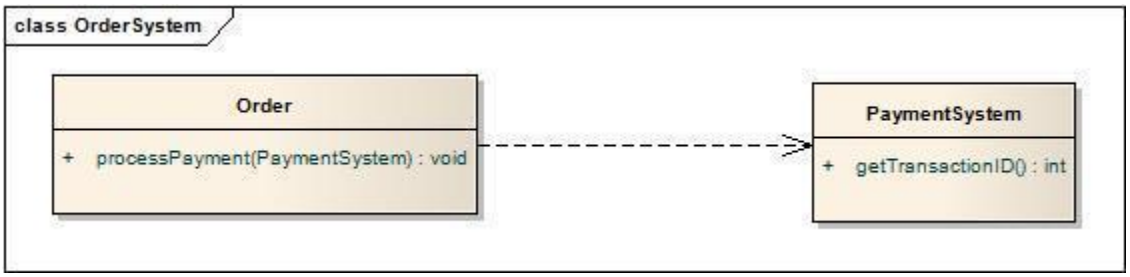
Realization – alternative

This is very straight forward implementation so hopefully there will ne no code provided. Unlike Generalization in this case the arrow is dashed.

Dependency

Dependency is a relationship that shows that a class is dependent on another class for its existence or implementation. Dependency relationship is shown as a dotted line with an arrow from source class to the dependent class.

In Java we can consider the dependency relationship if the source class has a reference to the dependent class directly or source class has methods through which the dependent objects are passed as a parameter or refers to the static operation's of the dependent class or source class has a local variable referring to the dependent class etc.



Dependency

The above diagram satisfies dependency relationship as the source class Order passes the PaymentSystem reference through the processPayment().

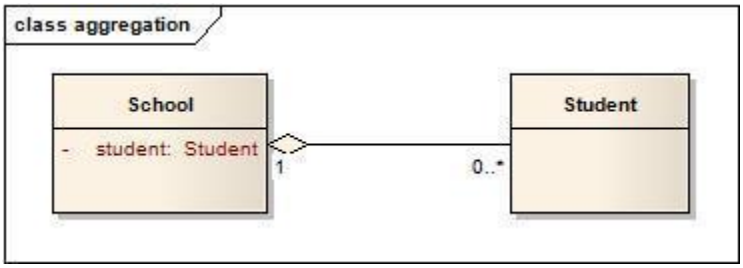
?

Code:

```
1  public class PaymentSystem {
2
3  }
```

```
4
5   public class Order {
6   public void processPayment(PaymentSystem ps){
7
8    }
9   }
```

Aggregation:
This shows “has a” relationship. It is a form of association relationship. This relationship highlights that a whole is made of its parts. So if a whole is destroyed the part still remains.
In UML this is represented through a hollow diamond with the diamond symbol pointing towards the whole.
In case of Java the aggregation follows the same structure as association. It is represented through the instance variables of a class.



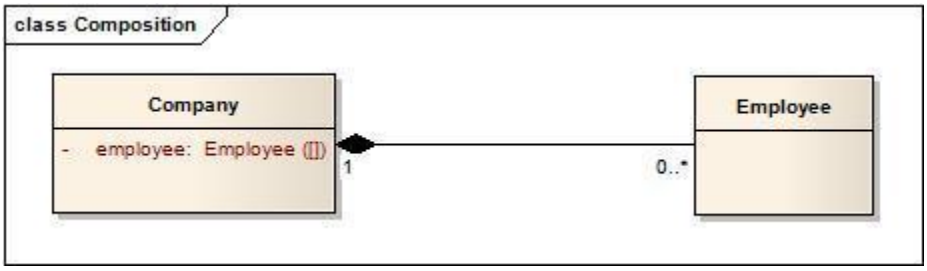
Aggregation
?

Code:

```
1   public class Student {
2
3   }
4
5   public class School {
6       private Student student;
7   }
```

In this case a student is a part of the School. However during design it is preferred to use association instead of aggregation as it is not a recommended option.

Composition:
This is again a whole or part relationship where if the whole is destroyed then the part cannot exist independently. Another important point about Composition is that the part at any point in time can have only one owner. E.g. A person can be an employee of one company at any point in time due to contractual obligations. That person cannot hold dual work authorisation. If the Company goes bankrupt the employee of this company does not exist and will be fired.
The composition is represented as a filled diamond with data flowing in single direction from the whole to the part.
The composition in Java is represented in the same form as aggregation with help of instance variables.



Composition
?

Code:

```
1   public class Employee {
2
3   }
4
5   public class Company {
6       private Employee[] employee;
7   }
```