Lecture 1

Object Oriented Design and Implementation Strategies

Objectives

The objective of this lecture is to introduce object-oriented design and implementation strategies. After this lecture, student will

- learn implementation strategies that are transforming design to code for object oriented principles (e.g., abstraction, encapsulation, decomposition, generalization, different types of inheritance, association, aggregation, and composition dependencies);
- explain different types of associations and learn how to create class definitions from UML class diagram to the Java code;
- lean how to apply design guidelines for modularity, separation of concerns, information hiding, and conceptual integrity to create a flexible, reusable, maintainable design
- learn the implementation tradeoff between cohesion and coupling.
- understand the SOLID principles that are the fundamental principles for object-oriented programming to be followed for adaptable changes to the class design and maintaintablity

Reference Reading

[1] Rajib Mall, "Fundamental of Software Engineering", 4th Edition, 2014
    - Chapter 8: Object Oriented Software Development

[2] Brahma Dathan and Sarnath Ramnath, "Object-Oriented Analysis, Design and Implementation" , An Integrated Approach, Second Edition, 2015, UTiCS, Springer.
    - Chapter 7: Design and Implementation

[3] https://www.visual-paradigm.com/guide/uml-unified-modeling-language/

[4] https://www.tutorialspoint.com/object_oriented_analysis_design/ooad_object_oriented_paradigm.htm

[5] https://www3.ntu.edu.sg/home/ehchua/programming/java/J3a_OOPBasics.html

[6] https://www.w3resource.com/java-exercises/index-interface.php

1. Introduction

In this lecture, the object oriented design model and the implementation strategies are discussed with practical example using Java technology. Students are expected to have basic Java programming knowledge. We have discussed the UML diagrams that represent different views for the software system in previous lecture. This lecture provides the implementation strategies for UML design models, transforming design to code.

The object design models come between the system design and implementation. After the designing stage, the implementation process gets start. Classes and their relationships which are developed during the object design; such designs are transformed into a particular programming language at the implementation stage. Generally, the task of converting an object design into source code is a straightforward process. But the feature such as the association of class diagrams in the design model is not possible to directly convert into programming language structures. Actually, most programming languages do not offer any paradigms to implement associations directly. This lecture, explores some of the significant features and talks about the several strategies that should be adopted for their implementation.

This lecture will explain concepts related to the implementation strategy, such as mapping design to code and illustrate how to transform class definition from class diagram and strategies for implementing unidirectional or bidirectional associations and how to create method from collaboration diagram, constraints and their implementation, statecharts and their implementation.

Firstly, let's recall our previous discussion on the object oriented concepts and design with UML diagram and then explain the implementation strategies for those uml design.

2. Object-Oriented Concepts, Design And Implementation

Basic concepts of the object oriented paradigm are the foundation elements to visualize a software application in the object model. The basic concepts and terminologies in object oriented systems are:

- Abstraction
  - Classes
  - Objects
- Encapsulation
- Inheritance
- Polymorphism

We discuss briefly about them with object oriented programming concept with java language.

2.1 Abstraction

Abstraction is a higher-level concept or a way of thinking when you start designing your application from the business requirement.

Abstraction means, simply, to **filter out an object's properties and operations**. For a business requirement, after you've made your decisions about what to include and what to exclude, is an abstraction of a system that you are going to build.

Abstraction is **a process of identifying essential entities (classes) and their characteristics (class members) and leaving irrelevant information from the business requirement to prepare a higher-level application design.**

Abstraction process includes finding nouns from the business requirement (the noun is the person, place, thing, or process) and identifying potential classes and their attributes and operations from the nouns.

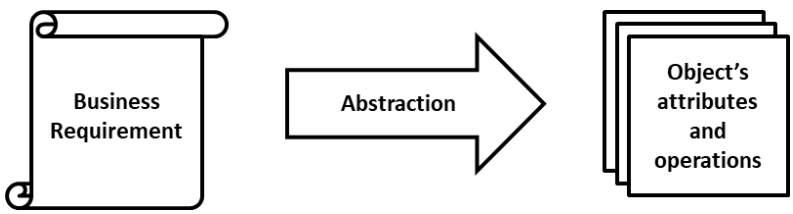


Figure 1.1: Abstraction concepts in object oriented design

2.2 Encapsulation

Encapsulation is **a technique to implement the abstraction in code**. Create ***classes*** and their ***attributes*** and ***operations*** with appropriate **access modifiers** to show functionalities and ***hide details and complexity***.

Encapsulation hides the data and implementation details show only the required members within a class, thus **hiding complexity from other code**. No other code needs to know about implementation detail and also can't modify the code of the class's data and methods. Encapsulation is also called **information hiding.** In the following section, we discuss encapsulation with example class implementation.

2.3 Classes and Objects

Object-oriented systems describe entities of the software system as **objects**. Objects are persons, places, or things that are relevant to the system we are analyzing. Typical objects may be customers, items, orders, and so on. **Each object** essentially **consists of some data** that is *private to the object* and **a set of functions** (termed as operations or methods) that *operate on those data*.

A class represents a **collection of objects having same characteristic properties** that exhibit common behavior. **A class** gives the **blueprint** or **description** of the **objects**

that can be created from it. **Creation of an object** as a member of a class is called ***instantiation***. Thus, **object is an instance of a class**. The constituents of a class are −

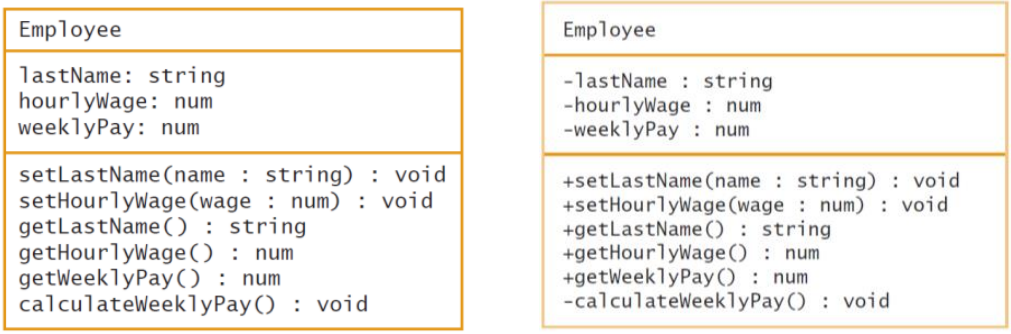
- A **set of attributes** for the objects. Attributes are referred as ***class data***.
- A **set of operations** that describe the ***behavior of the objects*** of the class. Operations are also referred as **functions** or **methods**.

2.3.1 A UML Class Diagram

A **UML class diagram** consists of a rectangle divided into three sections. The top section contains the name of the class, the middle section contains the names and data types of the attributes, and the bottom section contains the methods. For example, the **Employee** class shown in Figure 1.2 is the class diagram that contains **three attributes** (or data fields) and **six operations** or **methods**.

Attributes or data fields represent an employee's name, hourly pay rate, and weekly pay amount and operations or methods include two **set methods** that accept values from the outside world, three **get methods** that send data to the outside world and one **work method** that performs work within the class.

(1) The SET Methods: In the Employee class, the setLastName() and setHourlyWage() methods are known as **set methods** because their purpose is to set the values of data fields within the class. Each **accepts data from the outside and assigns it to a field** within the class.



(a) Employee class diagram in general form

(b) Employee class diagram with with public and private access specifiers

Figure 1.2: Employee class diagram

(2) The GET Methods: In the Employee class, three of the methods: getLastName(), getHourlyWage(), and getWeeklyPay() methods are called **get methods**. The purpose of a get method is to **return a value** to the world outside the class. The getLastName() method returns "string" type value of employee name, the getHourlyWage() method returns "number" type value of hourly wage, and the

getWeeklyPay() method returns “number” type value of the employee’s weekly pay amount.

(3) The WORK method: In the Employee class, calculateWeeklyPay() method is a work method within the class. It computes the weeklyPay value by multiplying hourlyWage by the work_week_hours. No values need to be passed into this method, and no value is returned from it because this method does not communicate with the outside world. Instead, this method is called only from within another method in the same class.

(4) The Access Specifier: An access specifier (or access modifier) is the defining the **type of access** (***public*** or ***private***) to the attribute or method of a class from the outside classes.

In object-oriented principles**, encapsulation** is the process of combining all of an object’s attributes and methods into a single package and **information hiding** is the concept that other classes should not alter an object’s attributes—only the methods of an object’s own class should have that privilege. Outside classes should only be allowed to make a request that an attribute be altered; then it is up to the class’s methods to determine whether the request is appropriate. Detailed workings of objects create within object-oriented programs can be hidden from outside programs and modules if you want them to be. Object-oriented programming defines these **encapsulation** and **information hiding principles** using the **access specifier**.

To prevent changing the class’s data fields from outside programs or methods, the object-oriented programmers usually specify that their attributes or data fields will have **private access**—that is, the data cannot be accessed by any method that is not part of the class. The **public access**—allows other programs and methods may use the methods to access to the class’s inside private data.

In UML class diagram, visibility symbols are used to define the access specifier. The UML defines four levels of visibility, namely ***public***, ***protected***, ***private*** and ***package***. **Visible symbols**:

- a plus sign (+) precedes the items are **public;**
- a hash sign (#) precedes those are **protected**,
- a minus sign (−) precedes the items are **private**;
- a (~) precedes those are **package**.

Figure 1.2 (a) is general Employee class and figure 1.2(b) with access specifier information.

Advantages of Encapsulation:

- Hides data and complexities.

- Restrict unauthorized access of data by allowing authorization before data access.
- Allow validation before setting data.
- Only the author of the class needs to understand the implementation, not others.
- Makes applications easy to maintain.

2.3.2 The Implementation of Class Diagram

Class diagram gives an overview of application software by depicting classes and their attribute, operations (methods) and relationships (dependencies, Generalization and Associations) among objects. This information is enough to create a basic class definition in an object-oriented language. The **class diagram** consists of ***classes***, ***interfaces***, ***super-classes***, ***methods*** and ***attributes***. These elements are sufficient to ***create a basic class definition of any object oriented languages like java.*** Mapping for methods and attributes signature to the code from the class diagram is straight forwarded.

The following example illustrates how to create class definitions from the class diagram. For the UML class diagram in Figure 1.1: Employee class diagram is implemented using java code is shown below:

```
public class Employee {

    private String lastName;
    private double hourlyWage;
    private double weeklyPay;

    // default constructor
    public Employee( ){
       this.hourlyWage = 10.00;
       calculateWeeklyPay();
    }
    // constructor for initial values
    public Employee(double rate, String name){
       this.lastName = name;
       setHourlyWage (rate);
       calculateWeeklyPay();
    }

    public void setLastName(string name){
       this.lastName = name;
    }

    public void setHourlyWage(double wage){
       float MAXWAGE = 70.00;
       float MINWAGE = 10.00;
```

```
        if (wage < MINWAGE){
              this.hourlyWage = MINWAGE;
        }
        else if (wage > MAXWAGE) {
               this.hourlyWage = MAXWAGE;
        }
        else {
               this.hourlyWage = wage;
        }
        calculateWeeklyPay();
    }

    public String getLastName(){
        return lastName;
    }

    public double getHourlyWage(){
        return hourlyWage;
    }

    public double getWeeklyPay(){
        return weeklyPay;
    }
    private void calculateWeeklyPay(){
        double  WORK_WEEK_HOURS = 40.0;
        this.weeklyPay  = this.hourlyWage * WORK_WEEK_HOURS;
    }
} // endClass
```

Mapping for Methods and Attributes Signature to the Code from the Class Diagram

- The **Employee class diagram** shown in figure 1.1(b) with **public** and **private access specifiers.**
- **Attributes** of the class are represented by fields in Java, with UML data types being translated into the ***appropriate Java equivalents where necessary***.
- **Operations** of the class are ***defined as methods in Java, with appropriate implementations.***
- A **default constructor** is one that requires **no arguments**. In OO languages, a default constructor is created automatically by the compiler for every class you write. Any constructor must have the ***same name as the class*** it constructs, and constructor methods ***cannot have a return type.***

- **A constructor** is defined to *initialize the values of the attributes*. Note that **constructors are often omitted from class diagrams**, but are **required in implementations of classes**.
- **The class diagram does not tell you what takes place inside the method**.
- When you create the Employee class, you **include method implementation details.**
- For example, in the setHourlyWage() method, the wage passed to the method is tested against minimum and maximum values, and is assigned to the class field hourlyWage only if it falls within the prescribed limits. If the wage is too low, the MINWAGE value is substituted, and if the wage is too high, the MAXWAGE value is substituted.
- The purpose of the **set method** is to set the values of data fields within the class.
- The purpose of a **get method** is to return a value to the world outside the class.

2.3.3 Object Creation or Instantiation

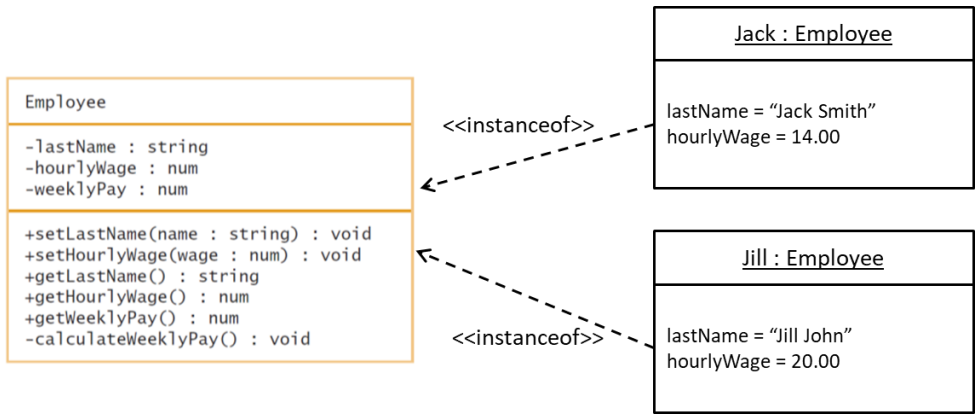


Figure 1.3: Employee class and its instances

An object is an instance of a particular class. Objects can be described as 'belonging to a class'. The class does not describe particular properties of individuals, but specifies the common features that are shared by all the objects in the group. For example, the relationship between objects and its class is represented in the UML is shown in figure 1.3 that shows objects representing employees in a company. That is objects creation or instantiation from a class.

2.3.4 The Implementation of Objects Instantiation from the Class

A TestEmployee class shown below is the test driver class that creates or instantiation an Employee object and sets the hourly wage value. The test driver class displays the weeklyPay value. Then a new value is assigned to hourlyWage and weeklyPay is displayed again. When Employee objects eventually are created, each

object will have its own lastName, hourlyWage and weeklyPay fields and have access to methods to get and set it. Now, compile and run this program to check how the object creation works. The execution result of the TestEmployee.java is shown in figure 1.4. As you can see from the output in Figure 1.4, the weeklyPay value has been recalculated even though it was never set directly by the client program.

```
public class TestEmployee {

   public static void main(String[] arg){
     double  LOW = 9.00;
     double  HIGH = 14.65;

     //declaration
     Employee   myGardener, myMaid;

     //Object creation or Object Instantiation
     myGardener = new Employee();
     myMaid = new Employee(22.50, "Parker");

     //setting values
     myGardener.setLastName("Greene");
     myGardener.setHourlyWage(LOW);

     //display the weeklypay
     System.out.println ("My gardener makes " + myGardener.getWeeklyPay() +" per week");
     System.out.println ("My maid makes " + myMaid.getWeeklyPay() +" per week");

     //setting new wage
     myGardener.setHourlyWage(HIGH);

     //getting values from class and display new weekly pay
     System.out.println("My gardener makes " + myGardener.getWeeklyPay() + " per week");
     System.out.println ("My maid makes " + myMaid.getWeeklyPay() +" per week");
   }
}
```

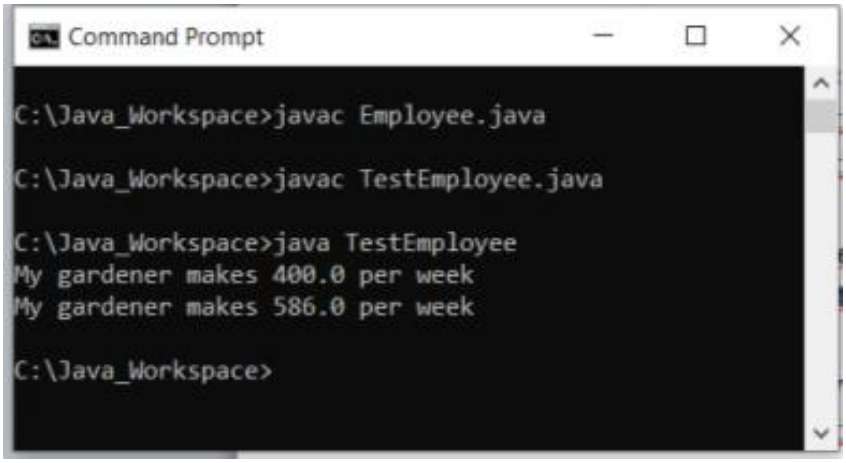


Figure 1.4: Execution of the TestEmployee class

2.3.5 Constructor Method

A constructor method, or a constructor, is a method that **establishes an object**. In OO languages, a default constructor is created automatically by the compiler for every class. Some define rules for constructor methods are as follows:

- A default constructor is one that requires **no arguments**.
- Any constructor methods have the **same name as the class** it constructs,
- Constructor methods **cannot have a return type**.
- Normally, declare constructors to be **public** so that other classes can instantiate objects that belong to the class.

Constructor methods are used for object's fields

- to provide initial values, for example, to set all numeric fields to zero by default.
- to hold some predefined default values, or
- to perform additional tasks when creating an instance of a class

For the Employee class, in implemented java code, the default constructor is Employee( ), that establishes one Employee object with the identifier provided. For example, if you want every Employee object to have a starting hourly wage of $10.00 as well as the correct weekly pay for that wage, then you could write the constructor for the Employee class as follows:

```
// default constructor
public Employee( ){
   this.hourlyWage = 10.00;
   calculateWeeklyPay();
```

```
}
```

Any Employee object instantiated will have an hourlyWage field value equal to 10.00, a weeklyPay field equal to $400.00, and a lastName field equal to the default value for strings. The another constructor with the parameter list is to set the predefined values specified by the user.

```
// constructor for initial values
public Employee(double rate, String name){
   this.lastName = name;
   this.hourlyWage = rate;
   calculateWeeklyPay();
}
```

In the test driver class, there are two employee objects: myGardener and myMaid. myGardener object use default constructor and myMaid object uses parameter constructor.

```
Employee   myGardener, myMaid;
myGardener = new Employee( );
myMaid = new Employee(22.50, "Parker");
```

2.3.6 Instance Methods

In the Employee class in above example, the method setLastName(string name) and setHourlyWage(double wage) method assigns a value to the lastName and hourlyWage fields for each separate Employee object create. Therefore, these s**et methods are called instance methods** because it **operates correctly yet differently** (using different values) **for each separate instance of the Employee class**.

In other words, if you create 100 Employees and assign lastName and hourlyWage to each of them, you need 100 storage locations in computer memory to store each unique lastName and hourlyWage fields.

2.3.7 Static Methods and Non-static Methods

In a class, there can be two types of methods: static methods and non-static methods.

(1) Static Methods

Static methods are also known as **class methods**. These methods **do not receive** a ***this*** reference **as an implicit parameter**. Typically, static methods include the word ***static*** in the method header. Figure 1.5: the Student class has one non-static methods: displayStudentMotto(). In most programming languages, use a static method with the class name, as in the following:

```
Student.displayStudentMotto();
```

In other words, **no object is necessary with a static method**, as shown shaded in TestStudent class in the code.

(2) Non-static Methods

Non-static methods are **instance methods**. These instance methods do receive a ***this*** reference to a specific object. Non-static methods are **used with an object created from a class.** In most programming languages, do not use any special word when you want a class member to be non-static. In other words, methods in a class are non-static instance methods by default. Student class has two static methods: setGradePointAverage(float gpa), getGradePointAverage(). In implementation code for Student class with static and non-static methods and TestStudent class using static methods and non-static methods as follows:

```
public class Student{

private float gradePointAverage;
public void setGradePointAverage(float gpa){
  this.gradePointAverage = gpa;
}
public float getGradePointAverage(){
  return this.gradePointAverage;
}
public static void displayStudentMotto(){
  System.out.print("Every student is an individual");
  System.out.println("in the pursuit of knowledge.");
  System.out.print("Every student strives to be");
  System.out.println("a literate, responsible citizen.");
}
}
```

```
public class TestStudent{

 public static void main(String[] arg){

   Student   S1 = new Student();
   Student   S2 = new Student();

   S1.setGradePointAverage(2.12);
   S2.setGradePointAverage(2.00);

   Student.displayStudentMotto();

}
}
```

Check Your Understanding on Classes and Objects

Q1. An instance method _____ .
(a) is static (b) receives a this reference (c) both of these (d) none of these

Q2. A static method is also known as a(n) _____ method.
(a) instance (b) public (c) private (d) class

Q3. By default, methods contained in a class are _____ methods.
(a) static (b) nonstatic (c) class (d) public

Q4. Assume you have created a class named MyClass, and that a working program contains the following statement:

```
output  MyClass.number
```

Which of the following do you know?

(a) number is a numeric field (b) number is a static field
(c) number is an instance variable (d) all of the above

Q5. Assume you have created an object named myObject and that a working program contains the following statement:

```
output myObject.getSize()
```

Which of the following do you know?

(a) size is a private numeric field (b) size is a static field
(c) size is a public instance variable (d) all of the above

Q6. When you instantiate an object, the automatically created method that is called is a(n) ---------.

(a) creator (b) initiator (c) constructor (d) architect

Q8. Every class has ----------.

(a) exactly one constructor (b) at least one constructor (c) at least two constructors
(d) a default constructor and a programmer-written constructor

Q7.Which of the following can be overloaded?

(a) constructors (b) instance methods (c) both of these (d) none of these

Q8. When you write a constructor that receives a parameter, _____.

(a) the parameter must be numeric
(b) the parameter must be used to set a data field
(c) the automatically created default constructor no longer exists
(d) the constructor body must be empty

Q9. Complete the following tasks:

(a) Write a java class named **Book** that holds a stock number, author, title, price, and number of pages for a book. Include methods to set and get the values for each data field. Create the class diagram and write the java code that defines the class.
(b) Write a test driver class that declares **two Book objects** and sets and displays their values.
(c) Write a test driver class that declares **an array of 10 Books**. Prompt the user for data for each of the Books, and then display all the values.

Q10. Complete the following tasks:

(a) Write a java class named GirlScout with fields that hold a name, troop number, and dues owed. Include get and set methods for each field. Include a static method that displays the Girl Scout motto ("To obey the Girl Scout law"). Include three overloaded constructors as follows:

- A default constructor that sets the name to "XXX" and the numeric fields to 0.
- A constructor that allows you to pass values for all three fields
- A constructor that allows you to pass a name and troop number but sets dues owed to 0.

Create the class diagram and write the pseudocode that defines the class.

b. Design an application that declares three GirlScout objects using a different constructor version with each object. Display each GirlScout's values. Then display the motto.

3. Relationship Between Classes

In object-oriented programming, **classes interact with each other** to accomplish one or more features of an application. You can **define the relationship between classes** while designing the classes of your application. There are three types of relationships in object-oriented programming based on how a class interacts with another class.

1. Inheritance (Is-A relation)
2. Composition (Has-A relation)
   - Composition
   - Aggregation
3. Association (Uses-A relation)

We discuss each relationship in following sections.

3.1 Inheritance Relationship Between Classes

Inheritance is a type of relationship between classes. Inheritance is a mechanism of **reusing the functionalities of one class into another related class**. It allows knowledge of a general category to more specific objects. The inheritance can only be used with related classes where they should have some common behaviors and perfectly substitutable.

Inheritance is referred to as **"is a" relationship**. For example, the company has two types of employee, employee who earns a commission as well as a weekly salary, called CommissionEmployee, and some employees who are paid by the hour and do not earn a weekly salary, called an HourlyEmployee. A commission employee "**is an**" employee and an hourly employee "**is also an**" employee of the company. These employees have employee number and weekly salary. So the Employee class with employee number and weekly salary data fields can be inherited to the ComissionEmployee class and HourlyEmployee class. In that way, object-oriented programming reuses the same attributes (or) data fields and functionalities of one class into another related class.

3.1.1 UML Notation for Inheritance Relationship

Figure 1.5 shows the UML inheritance relationship between Employee class and CommissionEmployee class with three fields (empNum, weeklySalary, commissionRate) and HourlyEmployee class with two fields (hoursWorked and hourlyRate).

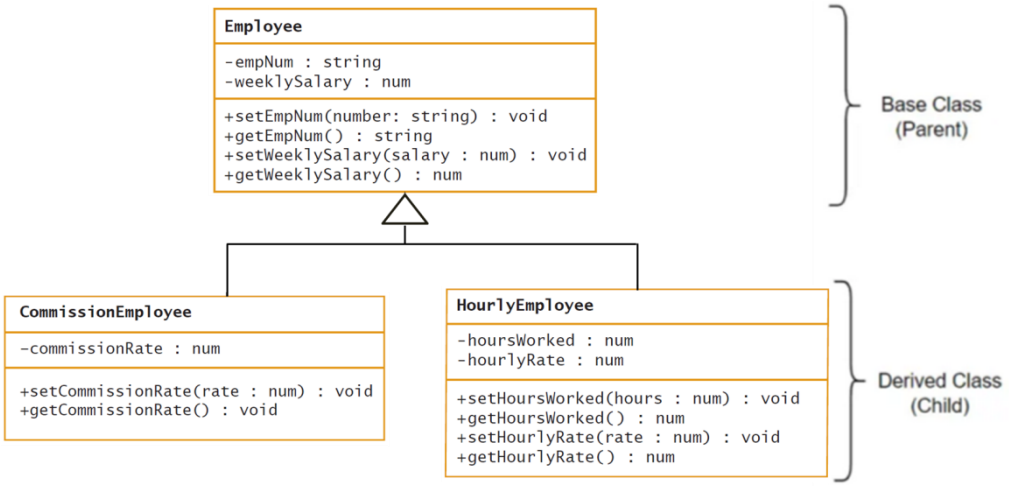


Figure 1.5: Inheritance relationship between classes

A class that is used as **a basis for inheritance**, like Employee class, is called **a base class**. When you create **a class that inherits from a base class** (such as CommissionEmployee, HourlyEmployee class), it is **a derived class** or **extended class**. A CommissionEmployee "**is an**" Employee—**not always the other way around**—so Employee **is the base class** and CommissionEmployee **is derived class**.

You can use the terms **superclass** and **subclass** as synonyms for base class and derived class. Thus, CommissionEmployee can be called a subclass of the Employee superclass. You can also use the terms **parent class** and **child class**. A CommissionEmployee **is a child to** the Employee **parent**.

A derived class can be further extended. In other words, a subclass can have a child of its own. The **entire list of parent classes** from which a child class is derived constitutes the **ancestors of the subclass**. A child inherits all the members of all its ancestors.

3.1.2 Implementation Strategies for Inheritance Relationship

The implementation strategies for inheritance relationship between classes in Figure 1.xx is shown below. In Java, use the "**extends**" keywords to inherit a class from another class. For example, the following CommissionEmployee class inherits from the `Employee` class in Java.

Example: Class Inheritance in Java

public class Employee {
private String empNum;
private float weeklySalary;
 	
public void setEmpNum(String number) {
	public class CommissionEmployee
	extends Employee {
	
	private float commissionRate;

```
        this.empNum = number;
    }
    public String getEmpNum( ){
        return this.empNum;
    }
    public void setWeeklySalary(float salary){
        this.weeklySalary = salary;
    }
    public float getWeeklySalary( ){
        return this.weeklySalary;
    }
}
```

```
    public void setCommissionRate (float
    rate)
        this.commissionRate = rate;
    }

    public float getCommissionRate( ) {
        return this.commissionRate;
    }
}
```

```
public class HourlyEmployee extends Employee {
    private float hoursWorked;
    private float hourlyRate;
    public void setHoursWorked(float hours){
        this.hoursWorked = hours;
        setWeeklySalary(hoursWorked * hourlyRate);  // call base class public set method
    }
    public float getHoursWorked( ){
        return this.hoursWorked;
    }
    public void setHourlyRate(float rate){
        this.hourlyRate = rate;
        setWeeklySalary(hoursWorked * hourlyRate);  // call base class public set method
    }
    public float getHourlyRate( ){
        return this.hourlyRate;
    }
}
```

In the above example, the Employee class is called the **base** class or the **parent** class, and the CommissionEmployee class is called the **derived** class or the **child** class.

The CommissionEmployee class and HourlyEmployee class inherit from the Employee class and so these two classes automatically **acquire all the public members of the Employee class**. It means even if the CommissionEmployee class does not include empNum, weeklySalary properties and set/get...() methods, an object of the CommissionEmployee class will have all the properties and methods of the Employee class along with its own members (commissionRate). See the executed results of following test driver TestInheritedMembers class, the CommissionEmployee object: salesperson can use empNum and weeklySalary data fields by using **public set methods**: salesperson.setEmpNum("222") and salesperson.setWeeklySalary(300.00) methods. The

HourlyEmployee class uses the base class public method, setWeeklySalary() inside its class to set the weeklysalary value.

Example: Inherited Members

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

      Employee   manager =  new Employee( );
      CommissionEmployee    salesperson  = new CommissionEmployee( );
      HourlyEmployee    parttimeworker = new HourlyEmployee( );

      manager.setEmpNum("111");
      manager.setWeeklySalary(700.00);

      salesperson.setEmpNum("222");              //call base class's public method
      salesperson.setWeeklySalary(300.00);     //call base class's public method
      salesperson.setCommissionRate(0.12);

      parttimeworker.setEmpNum("333");              //call base class's public method
      parttimeworker. setHourlyRate (20.12);
      parttimeworker. setHoursWorked (30.00);

      System.out.println("Manager: " + manager.getEmpNum()  +
                         " has weekly salary:  " + manager.getWeeklySalary() );

      System.out.println("Salesperson " +  salesperson.getEmpNum() +
                         " has weekly salary:  " +  salesperson.getWeeklySalary()  +
                          " with commission rate:  "  +  salesperson.getCommissionRate() );

      System.out.println("Part Time worker " +  parttimeworker.getEmpNum() +
                         " has weekly salary:  " +  parttimeworker.getWeeklySalary()  +
                          " with hourly rate:  "  +  parttimeworker. getHourlyRate() );
   }
}
```

Note that Java does not allow a class to inherit multiple classes. A class can only achieve multiple inheritances through interfaces. The following is the executed results of the test driver class, TestInheritedMembers.class.

```
Output:
Manager: 111  has weekly salary:
Salesperson: 222  has weekly salary:       with commission rate:  0.12
Part Time worker: 333  has weekly salary:       with hourly rate:  20.12
```

3.1.3 Role of Access Modifiers in Inheritance

Access modifiers play an **important role in inheritance**. Access modifiers of each **member in the base class** impact their **accessibility in the derived class**.

(1) Public Members

The public members of the base class are **accessible from the derived class** and also become **part of the derived class object**.

Example: Inheritance of Public Members

```
public class Employee {
      public String empNum;       { get; set; methods } // can be inherited
      public float weeklySalary;  { get; set; methods } // can be inherited
      ...
}
```

```
public class CommissionEmployee  extends Employee {
      ....

}
```

```
//Test driver class

CommissionEmployee  emp  =  new CommissionEmployee();
emp.empNum  =  "Bill-6-23";        // valid use of base class's public member
```

(2) Private Members

The private members of the base class **cannot be accessed directly** from the derived class and **cannot be part of the derived class object**.

Example: Inheritance of Private Members

```
public class Employee {
      private String empNum;       { get; set; methods} // cannot be inherited
      private float weeklySalary;  { get; set; methods} // cannot be inherited
      ...
}
```

```
public class CommissionEmployee  extends Employee {
      ....

}
```

```
 //Test driver class

 CommissionEmployee  emp  =  new CommissionEmployee();
 // base class's private member cannot be inherited
 emp.empNum  =  "Bill-6-23";      // Compile-time error
```

(3) Protected Members

The protected members of the base class can be **accessible from the derived class** but **cannot be a part of the derived class object**.

Example: Inheritance of Protected Members

```
public class Employee {
    protected String empNum;      { get; set; methods} // cannot be inherited
    protected float weeklySalary; { get; set; methods} // cannot be inherited
    ...
}
```

```
public class CommissionEmployee extends Employee {
    ....

    public String GetempNum() {
        // valid use of base class's protected member inside the derived class
            return this.empNum;
    }
}
```

```
    //Test driver class

    CommissionEmployee emp = new CommissionEmployee();
    emp.GetempNum( ); // Valid
    // base class's protected member cannot be used outside of derived class
    emp.empNum = "Bill-6-23"; // Compile-time error
```

Let consider access modifier with HourlyEmployee class given below. In the HourlyEmployee class, it set weeklySalary value by calling the public base class method setWeeklySalary() because weeklySalary remains ***private*** in Employee class. If weeklySalary is defined as ***protected*** access modifier in the base Employee class, the derived class can access the data fields inside the derived class but not accessible outside classes.

```
public class Employee {

    private String empNum;
    protected float weeklySalary;

    public void setEmpNum(String number)
    {
        this.empNum = number;
    }
    public String getEmpNum( ){
        return this.empNum;
    }
    public void setWeeklySalary(float
    salary){
```

```
public class HourlyEmployee extends
Employee {
    private float hoursWorked;
    private float hourlyRate;
  public void setHoursWorked(float hours){
    this.hoursWorked = hours;
    weeklySalary = hoursWorked *
    hourlyRate;
  }
  public float getHoursWorked( ){
    return this.hoursWorked;
```

```
        this.weeklySalary = salary;
    }
    public float getWeeklySalary( ){
        return this.weeklySalary;
    }
}
```

```
  }
  public void setHourlyRate(float rate){
    this.hourlyRate = rate;
    weeklySalary = hoursWorked *
    hourlyRate;
  }
  public float getHourlyRate( ){
    return  this.hourlyRate;
  }
}
```

```
//Test driver class

HourlyEmployee emp = new HourlyEmployee();
emp.setEmpNum("P1" );          // Valid
emp.setHourlyRate(22.00);
// base class's protected member cannot be used outside of derived class
emp.weeklySalary = 2500.00;   // Compile-time error,
```

3.1.4 Constructors Roles in Inheritance

Creating an object of the derived class will **first call the constructor of the base class** and then call the **constructor of the derived class**.

Example: Constructors in Inheritance

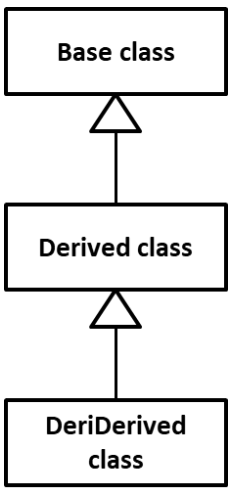
```
public class Employee {
    public Employee( ){
      System.out.println("Employee Constructor");
    }
    …
}
```

```
public class CommissionEmployee extends Employee {
    public  CommissionEmployee( ){
          System.out.println("Commission Employee Constructor");
    }
    …
}
```

```
//Test driver class

    CommissionEmployee emp = new CommissionEmployee();
```

```
//Output of Test driver class

    Employee Constructor
    Commission Employee Constructor
```

If there are multiple levels of inheritance then the constructor of the first base class will be called and then the second base class and so on. For example consider the following hierarchy of inheritance.




```
class Base {
   public Base() {
      System.out.println("Base");
   }
}
-----------------------------------------------------------------
class Derived extends Base {
   public Derived() {
      System.out.println("Derived");
   }
}
-----------------------------------------------------------------
class DeriDerived extends Derived {
   public DeriDerived() {
      System.out.println("DeriDerived");
   }
}
-----------------------------------------------------------------
public class Test {
   public static void main(String[] args) {
      Derived b = new DeriDerived();
   }
}
```

Output:

Base
Derived
DeriDerived

Explanation:

Whenever a class gets instantiated, the constructor of its base classes (the constructor of the ***root of the hierarchy*** gets ***executed first***) gets invoked before the constructor of the instantiated class.

3.1.5 Super Keyword

The keyword **super** refers to the **superclass**, which could be the **immediate parent** or its **ancestor**. Use the **super** keyword in the derived class to access the public members of the base class. When there is more than one constructor in the base class, to call specific constructor use "super" keyword. The keyword super allows the subclass to access superclass' methods and variables within the subclass' definition. For example, super() and super(argumentList) can be used invoke the superclass' constructor. If the subclass overrides a method inherited from its superclass, says getArea(), you can use

super.getArea() to invoke the superclass' version within the subclass definition. Similarly, if your subclass hides one of the superclass' variable, you can use super.variableName to refer to the hidden variable within the subclass definition. For example, the following calls the base class's parameterized constructor using the super key.

Example: super Keyword

```
public class Employee {
      private  String  empNum;
      private  float   weeklySalary;

    //constructors
    public  Employee ( ) {
         System.out.println("Employee  no-arg Constructor ");
    }
   //another constructor with parameter
    public   Employee (String empno) {
         this.empNum = empno;
         System.out.println("Employee:  "  +  empNum);
     }
   //another constructor with two parameters
    public   Employee (String  empno, float  wsalary) {
         this.empNum = empno;
         this.weeklySalary = wsalary;
         System.out.println("Employee:  " + empNum  +  "  with weeklySalary:  " + weeklySalary);
     }
          …
 }
```

```
 class  HourlyEmployee  extends  Employee  {

    //constructor
    public  HourlyEmployee() {
       super ( );              //call constructor of base class
    }
    public   HourlyEmployee(String empno) {
        super(empno);        // call Parameterized constructor of base class
    }
    public   HourlyEmployee(String  empno, float  salary){
      super(empno,  salary);   // call two parameterized constructor of base class
    }
 }
```

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

```
    HourlyEmployee emp = new  HourlyEmployee();
    HourlyEmployee emp1 =  new  HourlyEmployee("Emp 11");
    HourlyEmployee emp2 =  new  HourlyEmployee("Emp 22", 2300);
    …
  }
}
```

Output:

```
    Employee  no-arg Constructor
    Employee:  Emp 11
    Employee:  Emp 22    with weeklySalary:  2300
```

Note that:

- super(args), if it is used, must be the **first statement in the subclass' constructor**. If it is not used in the constructor, ***Java compiler automatically insert a super() statement to invoke the no-arg constructor of its immediate superclass***. This follows the fact that the parent must be born before the child can be born. You need to properly construct the superclasses before you can construct the subclass.
- If no constructor is defined in a class, Java compiler automatically create a no-argument (no-arg) constructor, that simply issues a super() call, as follows:

```
// If no constructor is defined in a class, compiler inserts this no-arg constructor
public ClassName () {
  super();  // call the superclass' no-arg constructor
}
```

- The default no-arg constructor will not be automatically generated, if one (or more) constructor was defined. In other words, you need to define no-arg constructor explicitly if other constructors were defined.
- If the immediate superclass does not have the default constructor (it defines some constructors but does not define a no-arg constructor), you will get a compilation error in doing a super() call. Note that Java compiler inserts a super() as the first statement in a constructor if there is no super(args).

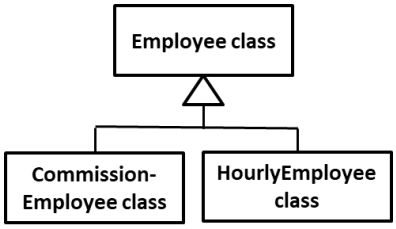
3.1.6 Object Initialization and Assignment in Inheritance

Creating an instance of **the derived class** and **assign** it to a variable of the **base class** or **derived** class. The instance's **properties** and **methods** are depending on the type of variable it is assigned to. Here, **a type** can be **a class** or **an interface**, or **an abstract class**. The following table list supported members based on a variable type and instance type.

Instance variable	Instance Type	Instance Members of
Base type	Base type	Base type
Base type	Derived type	Base type
Derived type	Derived type	Base and derived type

The following program demonstrates supported members based on the variable type:

Example: Object Creation and Assignment




```
public class Employee {
      public  String  empNum;
      public   float   weeklySalary;
      …constructors
      …get/set methods
}
```

```
public class   HourlyEmployee   extends   Employee {
      public   float  hoursWorked;
      public   float  hourlyRate;
      ..constructors
      ..get/set methods
}
```

```
public class CommissionEmployee  extends Employee {

    private   float    commissionRate;
    ..constructor
    ..get/set methods
}
```

//Test Driver Class

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

   //Instance variable = base type,  Instance type = base type, Instance members of = base type
      Employee    emp1 = new Employee ();
      emp1.empNum  = "emp-1";                 //valid
      emp1.weeklySalary =  700.00;            //valid
      emp1. hoursWorked;                     // not supported
      emp1. commissionRate;                  // not supported

   //Instance variable = base type,  Instance type = derived type, Instance members of = base type
      Employee   emp2 = new    CommissionEmployee();
      emp2. empNum =  "emp-2";                //valid
      emp2. weeklySalary = 650.00;            //valid
```

```
    emp2. commissionRate = 0.12;          // not supported

// Instance variable = derived type,  Instance type = derived type,
// Instance members of = base and derived type
    HourlyEmployee    hemp = new   HourlyEmployee();

    hemp. empNum  =  "Hemp-1";            //valid
    hemp. weeklySalary  = 650.00;         // valid

    hemp. hoursWorked  = 12.00;           //valid
    hemp.hourlyRate   = 20.00;            //valid

    //invalid, can't assign base type to derived type
    HourlyEmployee    emp  =  new  Employee();      //error
  }
}
```

In the above example, the type of emp2 is Employee, so it will ***only expose public properties*** of the Employee type even if an object type is the CommissionEmployee. However, the type of hemp is HourlyEmployee and so it exposes all the ***public properties of both classes***. Note that the **base** type object **cannot be assigned to** the **derived** type variable.

3.1.7 Object Type Conversion in Inheritance

The base type converts to the base class implicitly whereas the derived type must be converted to the base class explicitly using the as operator. In Java terms, object type conversion is called **upcasting** and **downcasting**.

(1) Upcasting a Subclass Instance to a Superclass Reference

Substituting a subclass instance for its superclass is called "upcasting". This is because, in a UML class diagram, subclass is often drawn below its superclass. **Upcasting is always safe** because ***a subclass instance possesses all the properties of its superclass and can do whatever its superclass can do***. The compiler checks for valid upcasting and issues error "incompatible types" otherwise.

For example,

```
Employee  emp1 =  new HourlyEmployee(12, 12.00);
// Compiler checks to ensure that Right-value is a subclass of Left-value.
Employee emp2 = new String();          // Compilation error: incompatible types
```

(2) Downcasting a Substituted Reference to Its Original Class

You can revert a **substituted instance back to a subclass reference**. This is called "downcasting". For example,

```
Employee emp1 = new HourlyEmployee(12, 12.00);    // upcast is safe
HourlyEmployee hemp1 = (HourlyEmployee) emp1;     // downcast needs the casting operator
```

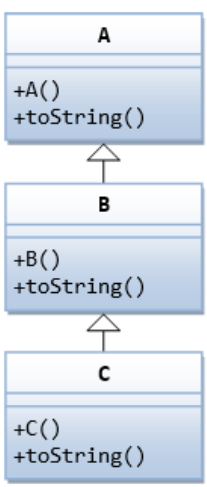
Downcasting requires **explicit type casting operator** in the form of ***prefix operator*** (new-type). **Downcasting is not always safe**, and throws a runtime ClassCastException if the instance to be downcasted does not belong to the correct subclass. A subclass object can be substituted for its superclass, but the reverse is not true.

(3) Casting Operator

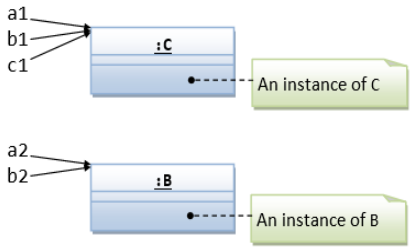
Compiler may not be able to detect error in explicit cast, which will be detected only at runtime. For example,

```
Employee emp = new Employee("Emp-5");
Company cmp = new Company( );
emp = cmp;      //compilation error: incompatible types (Company is not a subclass of Employee)
cmp = (Company) emp;  //runtime error:
                      //java.lang.ClassCastException: Company cannot be casted to Employee
```

Example: Type Conversion from Base to Derived object




```
public class A {
  public A() {  // Constructor
    System.out.println("Constructed an instance of A");
  }
  @Override
  public String toString() {
    return "This is A";
  }
}

-------------------------------------------------------------------------
public class B extends A {
  public B() {  // Constructor
    super();
    System.out.println("Constructed an instance of B");
  }
  @Override
  public String toString() {
    return "This is B";
  }
}
-------------------------------------------------------------------------
public class C extends B {
  public C() {  // Constructor
    super();
    System.out.println("Constructed an instance of C");
```




```
  }
  @Override
  public String toString() {
    return "This is C";
  }
}
```

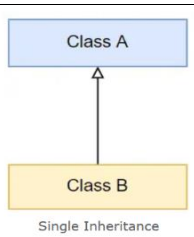
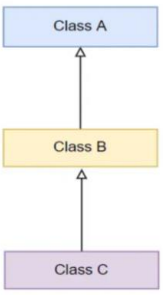
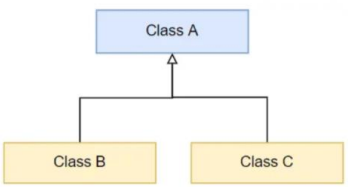
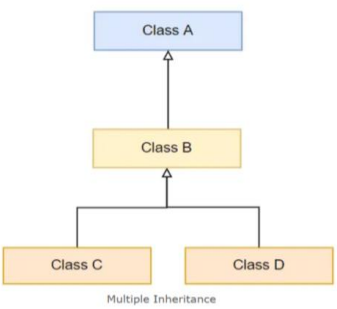
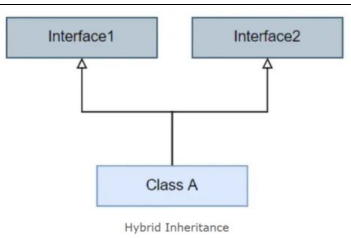
```
public class TestCasting {
  public static void main(String[] args) {
    A  a1 = new C( );            // upcast
    System.out.println(a1);      // run C's toString()
    B  b1 = (B) a1;              // downcast okay
    System.out.println(b1);      // run C's toString()
    C  c1 = (C) b1;              // downcast okay
    System.out.println(c1);      // run C's toString()

    A  a2 = new B();             // upcast
    System.out.println(a2);      // run B's toString()
    B  b2 = (B) a2;              // downcast okay
    C  c2 = (C) a2;              //compilation okay, but runtime error:
                                 //java.lang.ClassCastException: class B cannot be cast to class C
  }
}
```

Output:

```
    Constructed an instance of A
    Constructed an instance of B
    Constructed an instance of C
    This is C
    This is C
    This is C

    Constructed an instance of A
    Constructed an instance of B
    This is B
..runtime error: java.lang.ClassCastException: class B cannot be cast to class C
```

3.1.8 Types of Inheritance

There are different types of inheritance supported in Java based on how the classes are inherited.

(1)	**Single Inheritance** In a single inheritance, only one derived class inherits a single base class.	 Single Inheritance
(2)	**Multi-level Inheritance** In multi-level inheritance, a derived class inherits from a base class and then the same derived class becomes a base class for another derived class. Practically, there are no limits on the level of inheritance, but you **should avoid it.**	 Multi-level Inheritance
(3)	**Hierarchical Inheritance** In hierarchical inheritance, multiple derived classes inherit from a single base class.	 Hierarchical Inheritance
(4)	**Hybrid Inheritance** Hybrid inheritance is a combination of multi-level and hierarchical inheritance.	 Multiple Inheritance
(5)	**Multiple Inheritance** In multiple inheritance, a class inherits from multiple interfaces. Note that **Java does not support** deriving **multiple base classes**. Use ***interfaces*** for **multiple inheritance**.	 Hybrid Inheritance

3.1.9 Summary of Inheritance

- A class can inherit a single class only (i.e., single inheritance). It cannot inherit from multiple classes.
- A class can inherit (implement) one or more interfaces.
- Constructors or destructors cannot be inherited.

3.1.10 Exercise to Practice Inheritance - Superclass Person and its Subclasses

Suppose that we are required to model students and teachers in our application. We can define a **superclass** called Person to store common properties such as name and address, and **subclasses** Student and Teacher for their specific properties. For students, we need to maintain the courses taken and their respective grades; add a course with grade, print all courses taken and the average grade. Assume that a student takes no more than 30 courses for the entire program. For teachers, we need to maintain the courses taught currently, and able to add or remove a course taught. Assume that a teacher teaches not more than 5 courses concurrently.

We design the classes as follows.

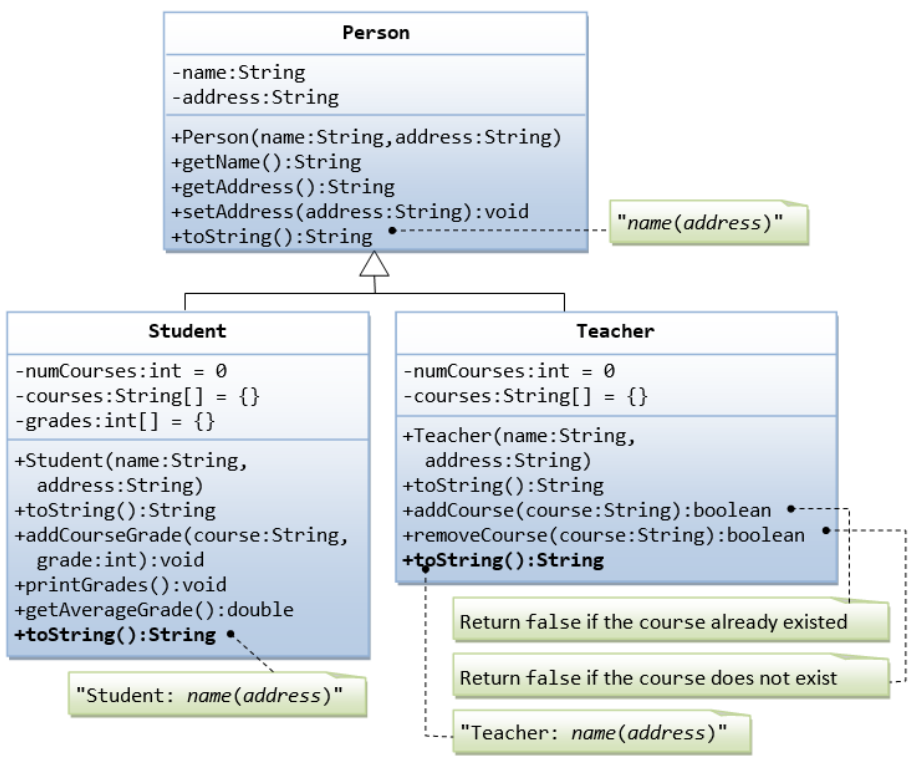


Figure 1.6: Practice Inheritance - Superclass Person and its Subclasses

The Superclass Person.java

```
/**
 * The superclass Person has name and address.
 */
public class Person {
  // private instance variables
  private String  name,  address;

  /** Constructs a Person instance with the given name and address */
  public Person(String  name, String  address) {
    this.name = name;
    this.address = address;
  }

  // Getters and Setters
  /** Returns the name */
  public String  getName() {
    return name;
  }
  /** Returns the address */
  public String  getAddress() {
    return address;
  }
  /** Sets the address */
  public void  setAddress(String address) {
    this.address = address;
  }

  /** Returns a self-descriptive string */
  @Override
  public String  toString() {
    return   name + "(" + address + ")";
  }
}
```

The Subclass Student.java

```
/**
 * The Student class, subclass of Person.
 */
public class Student extends Person {
  // private instance variables
  private  int   numCourses;   // number of courses taken so far
  private  String[]   courses;  // course codes
  private  int[]   grades;         // grade for the corresponding course codes
  private   static   final   int   MAX_COURSES = 30; // maximum number of courses
```

```
  /** Constructs a Student instance with the given name and address */
  public  Student (String  name, String  address) {
    super(name,  address);
    this.numCourses = 0;
    this.courses = new  String[MAX_COURSES];
    this.grades = new  int[MAX_COURSES];
  }

  /** Returns a self-descriptive string */
  @Override
  public  String  toString () {
    return  "Student:  "  +  super.toString();
  }

  /** Adds a course and its grade - No input validation */
  public  void  addCourseGrade (String course, int grade) {
    this.courses[numCourses] = course;
    this.grades[numCourses] = grade;
    ++numCourses;
  }

  /** Prints all courses taken and their grade */
  public  void  printGrades() {
    System.out.print(this);
    for (int i = 0; i < numCourses; ++i) {
      System.out.print(" " + courses[i] + ":" + grades[i]);
    }
    System.out.println();
  }

  /** Computes the average grade */
  public  double  getAverageGrade() {
    int sum = 0;
    for (int i = 0; i < numCourses; i++ ) {
      sum += grades[i];
    }
    return (double)sum/numCourses;
  }
}
```

The Subclass Teacher.java

```
/**
 * The Teacher class, subclass of Person.
 */
public class  Teacher  extends  Person {
  // private instance variables
```

```
  private int numCourses;  // number of courses taught currently
  private String[] courses; // course codes
  private static final int MAX_COURSES = 5; // maximum courses

  /** Constructs a Teacher instance with the given name and address */
  public Teacher (String name, String address) {
    super(name, address);
    this.numCourses = 0;
    this.courses = new String[MAX_COURSES];
  }

  /** Returns a self-descriptive string */
  @Override
  public String toString() {
    return "Teacher: " + super.toString();
  }

  /** Adds a course. Returns false if the course has already existed */
  public boolean addCourse (String course) {
    // Check if the course already in the course list
    for (int i = 0; i < numCourses; i++) {
      if (courses[i].equals(course)) return false;
    }
    courses[numCourses] = course;
    numCourses++;
    return true;
  }

  /** Removes a course. Returns false if the course cannot be found in the course list */
  public boolean removeCourse(String course) {
    boolean found = false;
    // Look for the course index
    int courseIndex = -1;  // need to initialize
    for (int i = 0; i < numCourses; i++) {
      if (courses[i].equals(course)) {
        courseIndex = i;
        found = true;
        break;
      }
    }
    if (found) {
      // Remove the course and re-arrange for courses array
      for (int i = courseIndex; i < numCourses-1; i++) {
        courses[i] = courses[i+1];
      }
      numCourses--;
      return true;
    } else {
```

```
      return false;
    }
  }
}
```

A Test Driver (TestPerson.java)

```
/**
 * A test driver for Person and its subclasses.
 */
public class  TestPerson {
  public  static  void  main(String[] args) {
    /* Test Student class */
    Student  s1 = new Student("Tan Ah Teck", "1 Happy Ave");
    s1.addCourseGrade("IM101", 97);
    s1.addCourseGrade("IM102", 68);
    s1.printGrades();
    System.out.println("Average is " + s1.getAverageGrade());

    /* Test Teacher class */
    Teacher   t1 = new Teacher("Paul Tan", "8 sunset way");
    System.out.println(t1);
    String[]   courses = {"IM101", "IM102", "IM101"};
    for (String course: courses) {
      if (t1.addCourse(course)) {
        System.out.println(course + " added");
      } else {
        System.out.println(course + " cannot be added");
      }
    }
    for (String course: courses) {
      if (t1.removeCourse(course)) {
        System.out.println(course + " removed");
      } else {
        System.out.println(course + " cannot be removed");
      }
    }
  }
}
```

Output of Test Driver

```
Student: Tan Ah Teck(1 Happy Ave) IM101:97 IM102:68
Average is 82.5
Teacher: Paul Tan(8 sunset way)
IM101 added
IM102 added
```

```
IM101 cannot be added
IM101 removed
IM102 removed
IM101 cannot be removed
```

Check Your Understanding on Inheritance Relationship Between Classes

Q1. Advantages of creating a class that inherits from another include all of the following except:

(a) You save time because subclasses are created automatically from those that come built-in as part of a programming language.

(b) You save time because you need not re-create the fields and methods in the original class.

(c) You reduce the chance of errors because the original class's methods have already been used and tested.

(d) You make it easier for anyone who has used the original class to understand the new class.

Q2. Employing inheritance reduces errors because _____.

(a) the new classes have access to fewer data fields

(b) the new classes have access to fewer methods

(c) you can copy and paste methods that you already created

(d) many of the methods you need have already been used and tested

Q3. A class that is used as a basis for inheritance is called a _____.

(a) derived class (b) subclass (c) child class (d) base class

Q4. Which of the following is true?

(a) A base class usually has more fields than its descendent.

(b) A child class can also be a parent class.

(c) A class's ancestors consist of its entire list of children.

(d) To be considered object oriented, a class must have a child.

Q5. Which of the following is true?

(a) A derived class inherits all the members of its ancestors.

(b) A derived class inherits only the public members of its ancestors.

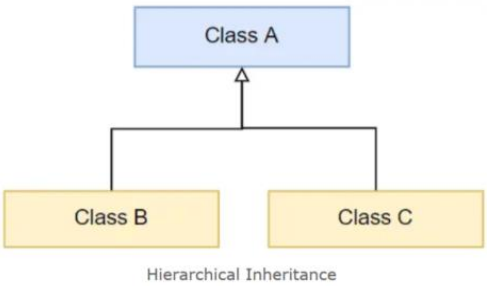
(c) A derived class inherits only the private members of its ancestors.

(d) A derived class inherits none of the members of its ancestors.

Q6. Which of the following is true?

(a) A class's data members are usually public. (b) A class's methods are usually public.
(c) Both of the above (d) none of the above

Q7. Which of the upcasting is valid for the UML class diagram?



Hierarchical Inheritance

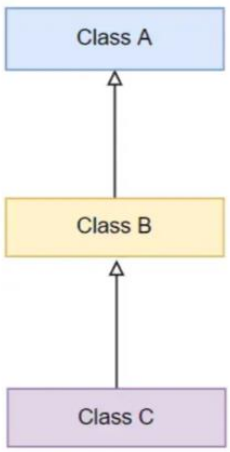
(a) ClassA obj-a = new ClassB();

(b) ClassB obj-b = new ClassA();

(c) ClassC obj-c = new ClassB();

(d) ClassB obj-b = new ClassC();

Q8. Which of the downcasting is invalid for the UML class diagram?



Multi-level Inheritance

ClassA obj-a = new ClassC();

(a) ClassB obj-b = (ClassB) obj-a;

(b) ClassC obj-c = (ClassC) obj-a;

(c) ClassC obj-c = (ClassB) obj-a;

(d) ClassB obj-b = (ClassC) obj-a;

Q9. Complete the following tasks:

(a) Write a java class named Book that holds a stock number, author, title, price, and number of pages for a book. Include methods to set and get the values for each data field. Also include a displayInfo() method that displays each of the Book's data fields with explanations.

(b) Write a java class named TextBook that is a child class of Book. Include a new data field for the grade level of the book. Override the Book class displayInfo() method so that you accommodate the new grade-level field.

(c) Write a test driver class that instantiates an object of each type and demonstrates all the methods.

Q10. Design java classes for following UML diagram:

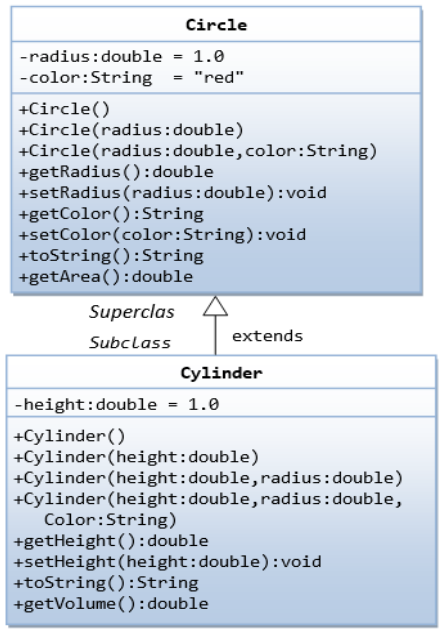
In this exercise, a subclass called Cylinder is derived from the superclass Circle.

The class Cylinder inherits all the member variables (radius and color) and methods (getRadius(), getArea(), among others) from its superclass Circle.

It further defines a variable called height, two public methods - getHeight() and getVolume() and its own constructors, as shown in figure.

The area of circle is pi * radius * radius.

The volume of the cylinder is circle area * height of the cylinder.



(a) Write a java class named Circle that holds a radius and color. Include methods that are shown in figure.

(b) Write a java class named Cylinder that is a child class of Circle. Include a new data field height of the cylinder. Show how the subclass Cylinder invokes the superclass' constructors (via super() and super(radius)) and inherits the variables and methods from the superclass Circle.

(c) Write a test driver class that instantiates an object of Cylinder using default constructor and another cylinder object with defined height and radius. Then display each cylinder object's radius, height, color and volume.

3.2 Polymorphism

Polymorphism is a Greek word that means **multiple forms** or **shapes**. You can use polymorphism if you want to have **multiple forms of one or more methods of a class with the same name**. Polymorphism can be achieved in two ways:

1. Compile-time Polymorphism (Method Overloading)
2. Runtime Polymorphism (Method Overriding)

3.2.1 Compile-Time Polymorphism (Method Overloading)

Compile-time polymorphism allows us to define more than one method with the same name but with different signatures. This is called **method overloading**. Method overloading is also known as **early binding** or **static binding** because which method to call is decided at compile time, early than the runtime.

Rules for Method Overloading:

(a) Method names should be the same but method signatures must be different. Either the number of parameters, type of parameters, or order of parameters must be different.
(b) The return type of the methods is not considered in the method overloading.
(c) Optional Parameters take precedence over implicit type conversion when deciding which method definition to bind.

3.2.2 Implementation Strategies for Compile-time Polymorphism

The following example in figure 1.7 demonstrates the method overloading by defining multiple constructors Circle() methods with a different number of parameters of the same type. The Circle class has three overloaded constructors - a default constructor with no argument, a constructor which takes a double argument for radius and a constructor which takes two arcuments: radius and color.

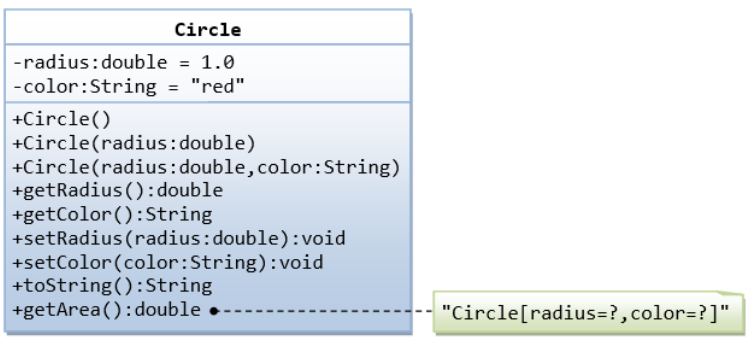


Figure 1.7: The Circle class with method overloading in constructor methods

Example: Method Overloading

```
public class Circle {
  // private instance variables
  private  double  radius;
  private  String   color;

  // Constructors
  public Circle() {
    this.radius = 1.0;
    this.color = "red";
    System.out.println("Construced a Circle with Circle()");  // for debugging
  }
  public Circle(double radius) {
    this.radius = radius;
    this.color = "red";
    System.out.println("Construced a Circle with Circle(radius)");  // for debugging
  }
  public Circle(double radius, String color) {
    this.radius = radius;
    this.color = color;
    System.out.println("Construced a Circle with Circle(radius, color)");  // for debugging
  }

  // public getters and setters for the private variables
  public double getRadius() {
    return this.radius;
  }
  public String getColor() {
    return this.color;
  }
  public void setRadius(double radius) {
    this.radius = radius;
  }
  public void setColor(String color) {
    this.color = color;
  }

  /** Returns a self-descriptive String */
  public String toString() {
    return "Circle[radius=" + radius + ",color=" + color + "]";
  }

  /** Returns the area of this Circle */
  public double getArea() {
    return radius * radius * Math.PI;
```

```
  }
}
```

Invoking Overloaded Methods

We can call the overloaded method by **passing the exact parameter it requires**. For example, if we want to invoke the Circle() method that set the default values for radius and color, we will pass no argument. Likewise, if we want to invoke the Circle(double radius, String color) method, we will pass double and String type argument.

Example: Invoking Overloading Method

```
public  class Test{
    public static void main(String[] args) {
        Circle  c1 = new Circle ( );                    //call no argument constructor
        Circle  c2 = new Circle (5.0, "green");      // call two parameterized constructor
        Circle   c3 = new Circle (5.0);              // call one parameterized constructor

        System.out.println("Circle 1 Radius  is " + c1.getRadius()  +  " , Color is " + c1.getColor()
                                    + " , Base area is " + c1.getArea() );
        // Circle 1 Radius is 1.0, Color is red, Base area is 3.141592653589793,

        System.out.println("Circle 2  Radius  is " + c2.getRadius()  +  " , Color is " + c2.getColor()
                                + " , Base area is " + c2.getArea() );
        // Circle 2 Radius is 5.0, Color is green, Base area is 3.141592653589793,

        System.out.println("Circle 3 Radius  is " + c3.getRadius()  +  " , Color is " + c3.getColor()
                                + " , Base area is " + c3.getArea() );
        // Circle 3 Radius is 5.0, Color is red, Base area is 3.141592653589793,
    }
}
```

Output:

```
Circle 1 Radius is 1.0, Color is red, Base area is 3.141592653589793
Circle 2 Radius is 5.0, Color is green, Base area is 3.141592653589793
Circle 3 Radius is 5.0, Color is red, Base area is 3.141592653589793,
```

3.2.3 Runtime Polymorphism (Method Overriding)

Run-time polymorphism is also known as **inheritance-based polymorphism** or **method overriding**. Inheritance allows you to inherit a base class into a derived class and all the public members of the base class automatically become members of the derived class. However, you can **redefine the base class's member in the derived class**

to provide a **different implementation than the base class**. This is called **method overriding** that also known as **runtime polymorphism**.

In **polymorphism**, a subclass inherits all the member variables and methods from its superclasses (the immediate parent and all its ancestors). It can use the inherited methods and variables as they are. It may also **override an inherited method by providing its own version**, or **hide an inherited variable by defining a variable of the same name**.

In the programming language, the compiler cannot know at compile time precisely which piece of codes is going to be executed at run-time (e.g., getArea() has different implementation for Rectangle and Triangle). To support the runtime polymorphism, object-oriented language like Java uses a mechanism called **dynamic binding** (or **late-binding** or **run-time binding**). When a method is invoked, the **code to be executed is only determined at run-time**. During the compilation, the compiler checks whether the method exists and performs type check on the arguments and return type, but does not know which piece of codes to execute at run-time. When a message is sent to an object to invoke a method, the object figures out which piece of codes to execute at run-time.

Rules for Overriding:

(a) A method, property, indexer, or event can be overridden in the derived class.
(b) Static methods cannot be overridden.
(c) Must use virtual keyword in the base class methods to indicate that the methods can be overridden.
(d) Must use the override keyword in the derived class to override the base class method.

Suppose that Shape class uses many kinds of **shapes**, such as **triangle**, **rectangle, circle** and so on. We should design a superclass called **Shape**, which defines the **public interfaces** (or behaviors) of all the shapes. For example, we would like all the shapes to have a method called **getArea**(), which returns the area of that particular shape. The **Shape** class can be written using the **abstract** class as follow.

3.2.4 Abstract Classes

An **abstract** method is a method with only signature (i.e., the method name, the list of arguments and the return type) without implementation (i.e., the method's body). Use the keyword abstract to declare an abstract method.

For example, in Figure 1.xx: the Shape class, we can declare abstract methods getArea(), draw(), etc, as follows:

```
abstract public class Shape {
        ……
        ……
```

```
        abstract public double getArea();
        abstract public double getPerimeter();
        abstract public void draw();
}
```

Implementation of these methods is NOT possible in the Shape class, as **the actual shape is not yet known**. (How to compute the area if the shape is not known?) Implementation of these abstract methods will be provided later once the actual shape is known. These abstract methods cannot be invoked because they have no implementation.

A class containing one or more abstract methods is called an abstract class. An abstract class must be declared with a class-modifier abstract. An abstract class CANNOT be instantiated, as its definition is not complete.

3.2.5 UML Notation for Abstract Class

An abstract class and methods are shown in ***italic***, in Figure 1.8.

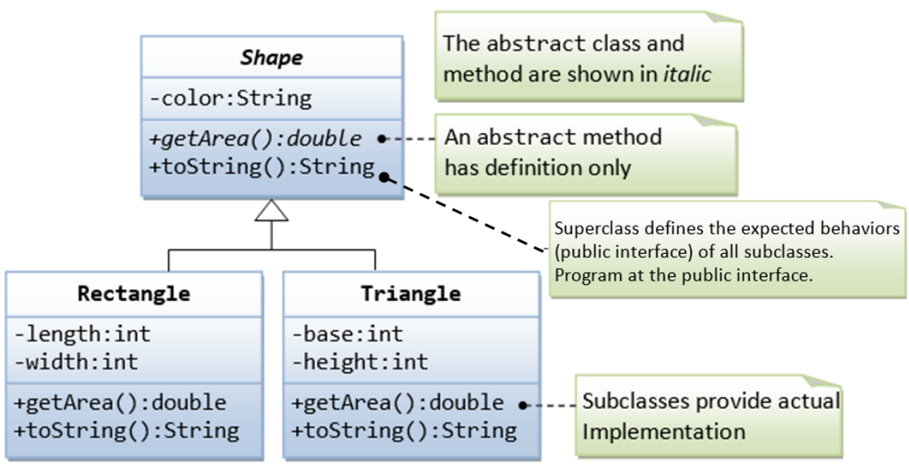


Figure 1.8: An abstract class and abstract methods of Shape class

3.2.6 Implementation Strategies for Runtime Polymorphism (Method Overriding) using Abstract Class

Let see our Shape class as an **abstract class**, containing an **abstract method** getArea() as follows:

Example: The abstract Superclass Shape.java

```
/**
 * This abstract superclass Shape contains an abstract method
 *  getArea(), to be implemented by its subclasses.
 */
```

```
abstract public class Shape {

  private String color;              // Private member variable

  /** Constructs a Shape instance with the given color */
  public Shape (String color) {
    this.color = color;
  }
  /** Returns a self-descriptive string */
  @Override
  public String toString() {
    return "Shape[color=" + color + "]";
  }
    /** All Shape's concrete subclasses must implement a method called getArea() */
  abstract public double getArea();
}
```

An abstract class is *incomplete* in its definition, since the implementation of its abstract methods is missing. Therefore, an abstract class ***cannot be instantiated***. In other words, you **cannot create instances from an abstract class** (otherwise, you will have an incomplete instance with missing method's body).

To use an abstract class, you have to **derive a subclass from the abstract class**. In the derived subclass, you have to **override the abstract methods and provide implementation to all the abstract methods.** The subclass derived is now complete, and can be instantiated. (If a subclass does not provide implementation to all the abstract methods of the superclass, the subclass remains abstract.) We derive subclasses, such as Triangle and Rectangle, from the superclass abstract Shape. The subclasses override the getArea() method inherited from the superclass, and provide the proper implementations for getArea() as follows:

Example: Subclass Rectangle.java

```
/**
 * The Rectangle class, subclass of Shape
 */
public class Rectangle extends Shape {

   private int length, width;   // Private member variables

  /** Constructs a Rectangle instance with the given color, length and width */
  public Rectangle(String color, int length, int width) {
    super(color);
```

```
    this.length = length;
    this.width = width;
  }

  /** Returns a self-descriptive string */
  @Override
  public String toString() {
    return "Rectangle[length=" + length + ",width=" + width + "," + super.toString() + "]";
  }

  /** Override the inherited getArea() to provide the proper implementation for rectangle */
  /**subclasses must implement an abstract method called getArea() */
  @Override
  public double getArea() {
    return length*width;
  }
}
```

Example: Subclass Triangle.java

```
/**
 * The Triangle class, subclass of Shape
 */
public class Triangle extends Shape {
  // Private member variables
  private int base, height;

  /** Constructs a Triangle instance with the given color, base and height */
  public Triangle(String color, int base, int height) {
    super(color);
    this.base = base;
    this.height = height;
  }
  /** Returns a self-descriptive string */
  @Override
  public String toString() {
    return "Triangle[base=" + base + ",height=" + height + "," + super.toString() + "]";
  }
  /** Override the inherited getArea() to provide the proper implementation for triangle */
  @Override
  public double getArea() {
    return 0.5*base*height;
  }
}
```

Example: A Test Driver (TestShape.java)

In our application, create instances of the subclasses such as **Triangle** and **Rectangle**, and **upcast** them and **assigned them to instances of superclass Shape** (so as to program and operate at the interface level), but you **cannot create instance of Shape**, which does not have implementation code. This is because the Shape class is meant to provide **a common interface to all its subclasses**, which are supposed to provide the actual implementation.

```
/**
 * A test driver for Shape and its subclasses
 */
public class TestShape {
  public static void main(String[] args) {

    Shape s1 = new Rectangle("red", 4, 5);      // Upcast
    System.out.println(s1);                     // Run Rectangle's toString()
    //Rectangle[length=4,width=5,Shape[color=red]]

     System.out.println("Area is " + s1.getArea());  // Run Rectangle's getArea()
    //Area is 20.0

    Shape s2 = new Triangle("blue", 4, 5);        // Upcast
    System.out.println(s2);                       // Run Triangle's toString()
    //Triangle[base=4,height=5,Shape[color=blue]]

    System.out.println("Area is " + s2.getArea());  // Run Triangle's getArea()
    //Area is 10.0

    // Cannot create instance of an abstract class
    Shape s3 = new Shape("green");
    //compilation error: Shape is abstract; cannot be instantiated
  }
}
```

The beauty of this code is that *all the references are from the superclass* (i.e., *programming at the interface level*). You could instantiate different subclass instance, and the code still works. You could extend your program easily by adding in more subclasses, such as Circle, Square, etc, with ease.

3.2.7 Summary of Abstract Class

In summary, an abstract class provides *a template for further development*. The **purpose of an abstract class is to provide a common interface** (or protocol, or

contract, or understanding, or naming convention) to all its subclasses. For example, in the abstract class Shape, you can define abstract methods such as getArea() and draw(). No implementation is possible because the actual shape is not known. However, by specifying the signature of the abstract methods, **all the subclasses are *forced* to use these methods' signature.** The subclasses could provide the proper implementations.

Coupled with polymorphism, you can **upcast** subclass instances to **Shape**, and program at the **Shape** level, i,e., program at the **interface**. The separation of interface and implementation enables **better software design**, and **ease in expansion**. For example, Shape defines a method called getArea(), which all the subclasses must provide the correct implementation. You can ask for a getArea() from any subclasses of Shape, the correct area will be computed. Furthermore, you application can be extended easily to accommodate new shapes (such as Circle or Square) by deriving more subclasses.

Rule of Thumb: Program at the interface, not at the implementation. (That is, make references at the superclass; substitute with subclass instances; and invoke methods defined in the superclass only.)

Substitutability

A subclass possesses all the attributes and operations of its superclass (because a subclass inherited all attributes and operations from its superclass). This means that a subclass object can do whatever its superclass can do. As a result, we can substitute a subclass instance when a superclass instance is expected, and everything shall work fine. This is called **substitutability.**

Notes:

- An abstract method **cannot be declared final**, as **final method cannot be overridden.** An abstract method, on the other hand, must be overridden in a descendant before it can be used.
- An abstract method **cannot be private** (which generates a compilation error). This is because private methods are not visible to the subclass and thus cannot be overridden.

3.2.8 Interface Class and the Java's Interface

A Java interface is a *100% abstract superclass* which define a set of methods its subclasses must support. **An interface contains only public *abstract methods*** (methods with signature and no implementation) and **possibly *constants*** (public static final variables). You have to use the **keyword** "**interface**" to define an interface (instead of keyword "class" for normal classes). The **keyword public and abstract are not needed for its abstract methods as they are mandatory**.

(JDK 8 introduces default and static methods in the interface. JDK 9 introduces private methods in the interface. These will not be covered in this article.)

Similar to an ***abstract*** superclass, an ***interface*** **cannot be instantiated**. You have to **create a "subclass" that implements an interface**, and provide the actual implementation of all the abstract methods. Unlike a normal class, where you use the keyword "**extends**" to derive a subclass. For **interface**, we use the keyword "**implements**" to derive a subclass.

An **interface** is a *contract* for what the classes can do. It, however, does not specify how the classes should do it. An interface provides a *form*, a *protocol*, a *standard*, a *contract*, a *specification*, a set of *rules*, an *interface*, for all objects that implement it. It is a *specification* and *rules* that any object implementing it agrees to follow.

In **Java**, **abstract** class and **interface** are used to separate the public *interface* of a class from its *implementation* so as to allow the programmer to program at the *interface* instead of the various *implementation*.

Interface Naming Convention: Use an adjective (typically ends with "able") consisting of one or more words. Each word shall be initial capitalized (camel-case). For example, **Serializable, Extenalizable, Movable, Clonable, Runnable, etc.**

3.2.9 UML Notations for *Interface* Formal Syntax

The formal syntax for declaring ***interface*** is:

```
[public|protected|package] interface interfaceName
[extends superInterfaceName] {

  // constants
  static final ...;

  // public abstract methods' signature
  ...
}
```

All methods in an **interface** shall be public and abstract (**default**). You cannot use other access modifier such as private, protected and default, or modifiers such as static, final.

All fields shall be public, static and final (**default**).

An interface may "extends" from a super-interface.

UML Notation: The UML notation uses a **solid-line arrow linking** the subclass to a concrete or abstract superclass, and **dashed-line arrow** to an interface as illustrated. **Abstract class** and **abstract method** are shown in ***italics***.

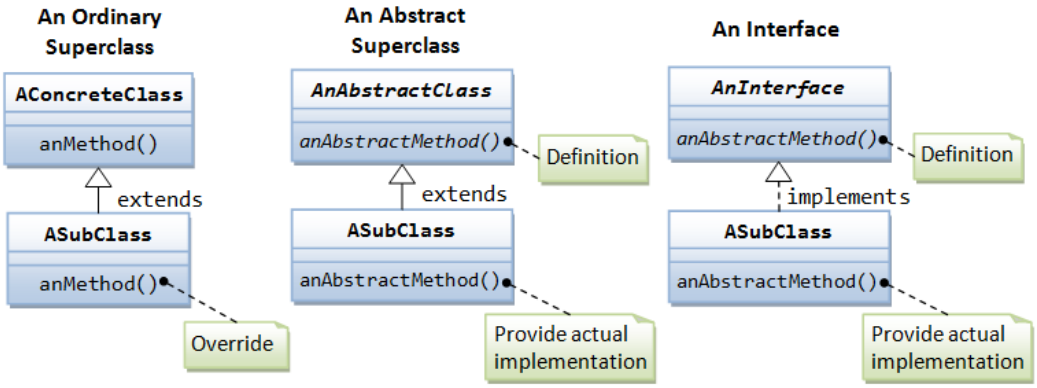


Figure 1.9: Normal class, Abstract class, abstract method, interface syntax

Example: Interface - Shape Interface and its Implementations

We can re-write the abstract superclass Shape into an interface, containing only abstract methods, as follows:

Abstract classes, **Interfaces** and **abstract methods** are shown in ***italics***. Implementation of **interface** is marked by a **dash-arrow leading from the subclasses to the interface,** shown in Figure 1.10, the interface Shape class and its inherited classes.

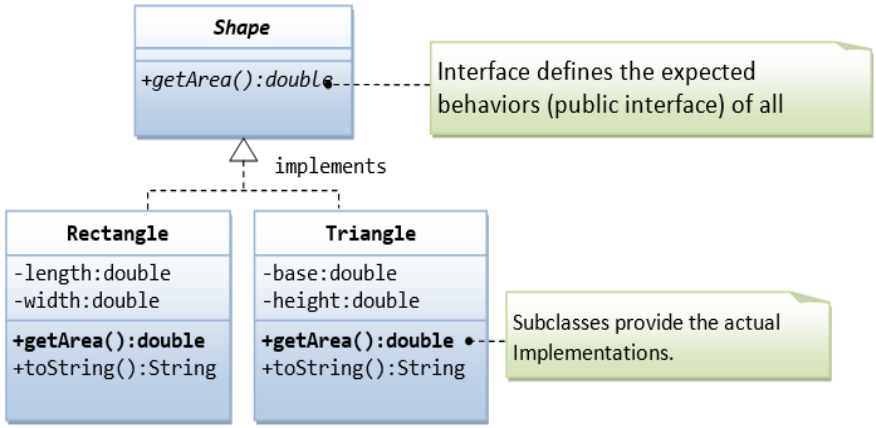


Figure 1.10: An interface of Shape class and its implementation to Rectangle and Triangle class

```
/**
 * The interface Shape specifies the behaviors
 *   of this implementations subclasses.
 */

public  interface  Shape {          // Use keyword "interface" instead of "class"
  // List of public abstract methods to be implemented by its subclasses
  // All methods in interface are "public  abstract".
  // "protected", "private" and "package" methods are NOT allowed.
```

```
   double getArea();            //public abstract double getArea( )

}

/**
 * The subclass Rectangle needs to implement all the abstract methods in Shape
 */
public class Rectangle implements Shape {
              // using keyword "implements" instead of "extends"

 // Private member variables
  private int length, width;

  /** Constructs a Rectangle instance with the given length and width */
  public  Rectangle(int length, int width) {
    this.length = length;
    this.width = width;
  }

  /** Returns a self-descriptive string */
  @Override
  public  String  toString() {
    return "Rectangle[length=" + length + ",width=" + width + "]";
  }

  // Need to implement all the abstract methods defined in the interface
  /** Returns the area of this rectangle */
  @Override
  public  double  getArea() {
    return length * width;
  }
}

/**
 * The subclass Triangle need to implement all the abstract methods in Shape
 */
public class Triangle implements Shape {
  // Private member variables
  private int base, height;

  /** Constructs a Triangle instance with the given base and height */
  public Triangle(int base, int height) {
    this.base = base;
    this.height = height;
  }

  /** Returns a self-descriptive string */
  @Override
  public String toString() {
    return "Triangle[base=" + base + ",height=" + height + "]";
  }
```

```
  // Need to implement all the abstract methods defined in the interface
  /** Returns the area of this triangle */


 @Override
  public double getArea() {
        return     0.5 * base * height;
  }
}
```

A test driver class is as follows:

```
public class TestShape {
  public static void main(String[] args) {
    Shape s1 = new Rectangle(1, 2);     // upcast
    System.out.println(s1);                  //Rectangle[length=1,width=2]
    System.out.println("Area is " + s1.getArea());        //Area is 2.0

    Shape s2 = new Triangle(3, 4);  // upcast
    System.out.println(s2);              //Triangle[base=3,height=4]
    System.out.println("Area is " + s2.getArea());         //Area is 6.0

    // Cannot create instance of an interface
    Shape s3 = new Shape("green");
    //compilation error: Shape is abstract; cannot be instantiated
  }
}
```

3.2.10 Implementing Multiple Interfaces

As mentioned, Java supports only *single inheritance*. That is, a subclass can be derived from one and only one superclass. Java does not support *multiple inheritances* to avoid inheriting conflicting properties from multiple super classes. But Java supports the multiple inheritances; here **a subclass can extend any number of interfaces**. Do not forget to **implement all of the methods of all the Interfaces**, otherwise compilation will fail!

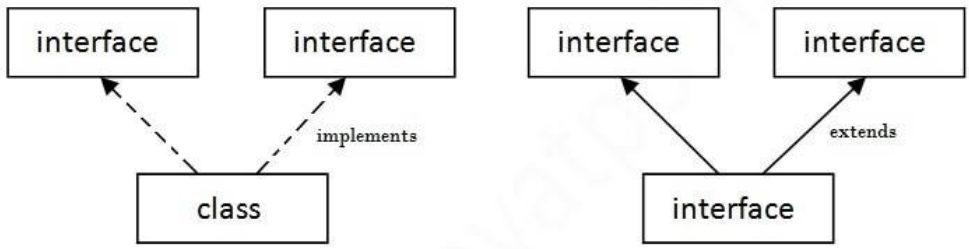


Figure 1.11: Multiple Inheritances of a class in Java

A subclass, however, can implement more than one interface. This is permitted in Java as an interface merely defines the abstract methods without the actual implementations and less likely leads to inheriting conflicting properties from multiple interfaces. In other words, **Java indirectly supports multiple inheritances via implementing multiple interfaces. Any class that implements multiple interfaces must provide an implementation for every method defined in each of the interfaces it implements.**

For example,

```
public class Circle extends Shape implements Movable, Resizable {
    // extends one superclass but implements multiple interfaces
  .....
}
```

Some features of Interfaces

- You can place variables within an Interface, although it won't be a sensible decision as Classes are not bound to have the same variable. In short**, avoid placing variables in interface!**
- **All variables and methods in an Interface are public**, even if you leave out the public keyword.
- An Interface **cannot specify the implementation of a particular method**. It's up to the Classes to implement it.
- If a Class implements multiple Interfaces, then there is a **remote chance of method signature overlap.** Since **Java does not allow multiple methods of the exact same signature, this can lead to problems.**

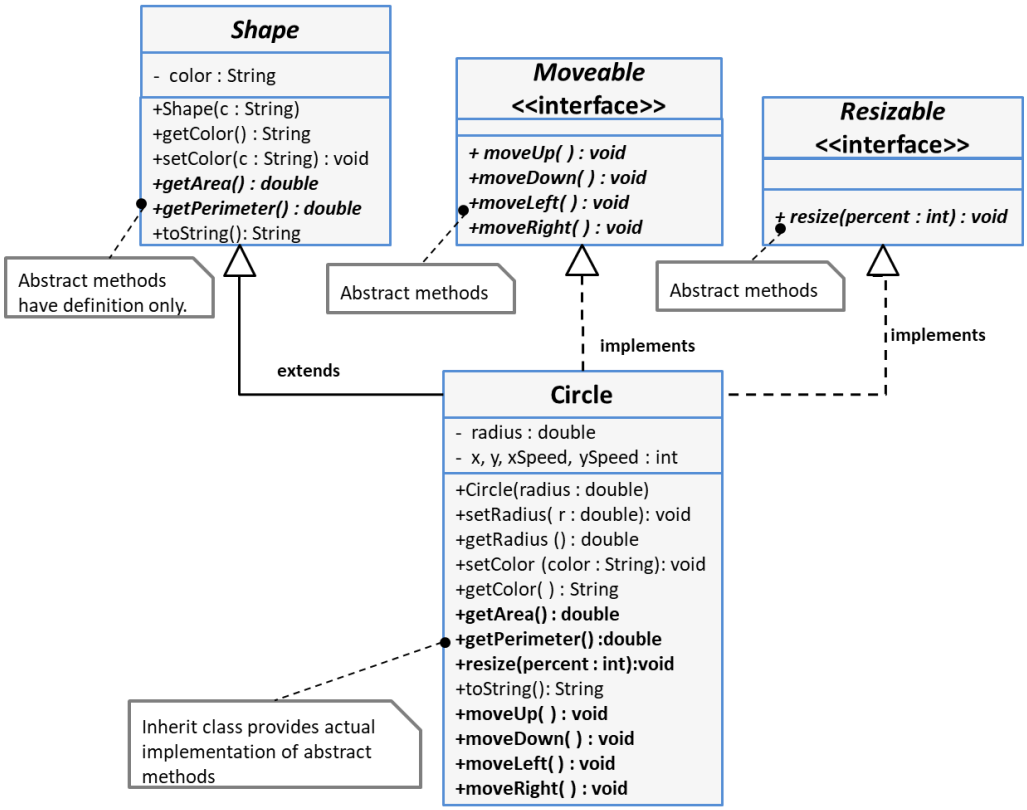


Figure 1.12: Multiple inheritance of a Circle class

Example: Multiple Inheritances

The Abstract Shape Class

```
/**
 * This abstract superclass Shape contains an abstract method
 *  getArea()  and  getPerimeter() to be implemented by its subclasses.
 */
abstract public  class  Shape {

  protected  String  color;              // Protected member variable

  /** Constructs a Shape instance with the given color */
  public  Shape (String color) {
    this.color = color;
  }
// public getters and setters for the protected variables
  public String getColor( ) {
    return this.color;
  }
  public void setColor(String color) {
    this.color = color;
```

```
  }
  /** Returns a self-descriptive string */
  @Override
  public  String  toString() {
    return "Shape[color=" + color + "]";
  }
    /** All Shape's concrete subclasses must implement a method called getArea() */
  abstract  public  double  getArea();
  abstract  public  double  getPerimeter();

}
```

The Interface Moveable Class

```
/**
 * The Movable interface defines a list of public abstract methods
 *   to be implemented by its subclasses
 */
public interface Movable {
  // use keyword "interface" (instead of "class") to define an interface
  // An interface defines a list of public abstract methods to be implemented by subclasses

  public void moveUp();                  // "public" and "abstract" optional
  public void moveDown();
  public void moveLeft();
  public void moveRight();

}
```

The Interface Resizable Class

```
/**
 * The Movable interface defines a list of public abstract methods
 *   to be implemented by its subclasses
 */
public interface Resible {
  // use keyword "interface" (instead of "class") to define an interface
  // An interface defines a list of public abstract methods to be implemented by subclasses

  public void  resize(int   percent);          // "public" and "abstract" optional

}
```

The Circle class

```
/**
 * This subclass  Circle extends the Abstract Shape class,
 * implements  two interface classes: Movable and Resizable
 * The circle class implements abstract methods from its super classes.
 */
```

```
public class Circle extends Shape implements Movable, Resizable {
  // extends one superclass but implements multiple interfaces
 // private instance variables
  private double radius;
  private int x, y, xSpeed, ySpeed;

  // Constructors
  public Circle(double radius) {
    this.radius = radius;
    super("red");
    x = 10;  y = 5;  xSpeed = 0.5;  ySpeed = 0.3;
    System.out.println("Construced a Circle with Circle(radius)");  // for debugging
  }

  // public getters and setters for the private variables
  public double getRadius() {
    return this.radius;
  }
  public String  getColor() {
    return super.getColor();
  }
  public void setRadius(double radius) {
    this.radius = radius;
  }
  public void setColor(String color) {
    super.setColor(color);
  }

  /** Returns a self-descriptive String */
  public String toString() {
    return "Circle[radius=" + radius + " , color= " + super.getColor( ) +
                 " , (x,y) = " +  x  +  "  ,  "   +   y  +  " ]";
  }

  /** All Shape's concrete subclasses must implement a method called getArea() */
 /** Returns the area of this Circle */
   @Override
  public double getArea() {
    return    radius * radius * Math.PI;
  }
/** All Shape's concrete subclasses must implement a method called getPeremeter() */
  /** Returns the perimeter of this Circle */
 @Override
  public double  getPerimeter() {
    return   2  * radius  * Math.PI;
  }

  /** All interface Moveable methods must implement in a subclass */
/**  Implement all abstract methods declared in the interface Movable */
  @Override
  public void moveUp(){
```

```
        y  - = ySpeed;
    }
   @Override
    public void moveDown(){
        y  + = ySpeed;
    }
    @Override
    public void moveLeft(){
        x  - =  xSpeed;
    }
    @Override
    public void moveRight(){
        x  + =  xSpeed;
    }

   /** Implement all abstract methods declared in the interface Resizable  */
    @Override
    public void  resize(int   percent){
        radius  * =  percent / 100.0;
    }
}
```

The Test Deriver Class

```
/**
 * This Test class  shows how Circle class multiple inheritances from superclasses.
 * /
public class TestMultipleInterfaces {
   public static void main(String[] args) {
      Shape  c1 = new Circle(3.0);    // upcast
      System.out.println(c1);        // Circle[radius= 3.0 , color= red , (x,y) = 10 , 5 ]
      c1.moveLeft();
      System.out.println(c1);       // Circle[radius= 3.0 , color= red , (x,y) = 9.5 , 5 ]
      c1.moveDown();
      System.out.println(c1);       // Circle[radius= 3.0 , color= red , (x,y) = 9.5 , 5.3 ]
      c1.moveRight();
      System.out.println(c1);      // Circle[radius= 3.0 , color= red , (x,y) = 10 , 5.3 ]
      c1.resize(20);
      System.out.println(c1);      // Circle[radius= 0.6 , color= red , (x,y) = 10 , 5.3 ]
}
```

3.2.11 Interface vs. Abstract Superclass

Which is a better design: interface or abstract superclass? There is no clear answer.

Use abstract superclass if there is **a clear class hierarchy**. Abstract class can contain partial implementation (such as instance variables and methods). Interface cannot

contain any implementation, but merely defines the behaviors. As an example, Java's thread can be built using interface Runnable or superclass Thread.

3.2.12 Exercise to Practices Polymorphism, Abstract and Interface

Rewrite the superclass Shape and its subclasses Circle, Rectangle and Square, as shown in the class diagram.

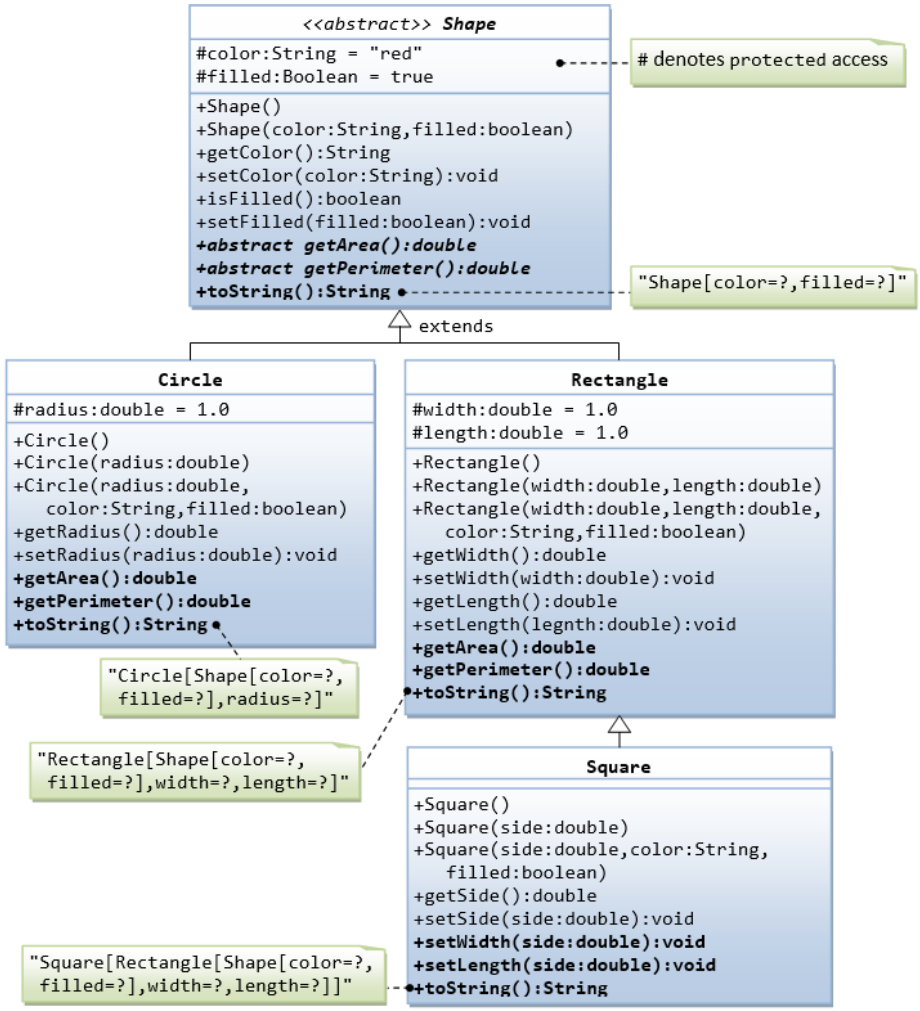


Figure 1.13: The Circle and Cylinder class,
practicing the Polymorphism, Abstract and Interface

Shape is an abstract class containing two abstract methods: getArea() and getPerimeter(), where its concrete subclasses must provide its implementation. All instance variables shall have protected access, i.e., accessible by its subclasses and classes in the same package. Mark all the overridden methods with annotation @Override.

In this exercise, Shape shall be defined as an abstract class, which contains:

- Two protected instance variables color(String) and filled(boolean). The protected variables can be accessed by its subclasses and classes in the same package. They are denoted with a '#' sign in the class diagram.
- Getter and setter for all the instance variables, and toString().
- Two abstract methods getArea() and getPerimeter() (shown in italics in the class diagram).

The subclasses Circle and Rectangle shall ***override*** the abstract methods getArea() and getPerimeter() and provide the proper implementation. They also ***override*** the toString().

The Abstract class: Shape

```
/**
 * This abstract superclass Shape contains an abstract method
 *  getArea(), to be implemented by its subclasses.
 */

abstract public class Shape {

  protected  String  color;              // Protected member variable
  protected  Boolean  filled;

  // Constructors (overloaded)
  /** Constructs a Shape instance with default value for color and filled */
  public  Shape ( ) {                    // 1st (default) constructor
    this.color = "red";
    this.filled = true;
  }

  /** Constructs a Shape instance with the given color and filled*/
  public  Shape (String color, Boolean filled) {              // 2nd constructor
    this.color = color;
    this.filled  = filled;
  }

  // public getters and setters for the protected variables
  public String getColor( ) {
    return this.color;
  }
  public void setColor(String color) {
    this.color = color;
  }
  public Boolean isFilled( ) {
    return this.filled;
```

```
  }
  public void setFilled(Boolean filled) {
    this.filled = filled;
  }

    /** All Shape's concrete subclasses must implement a method called getArea() */
  abstract public double getArea();
  abstract public double getPerimeter();

/** Returns a self-descriptive string */
  @Override
  public String toString() {
    return "Shape[color=" + this.color + ", filled=" + this.filled + " ]";
  }
}
```

The subclass: Circle

```
/**
 * The subclass Circle class models a circle with a radius,
*   the color and filled are inherited from Shape class.
 */

public class Circle extends Shape{
  // protected instance variables
  protected double radius;

  // Constructors overloaded
  public Circle() {
    super();                  //setting default color=red and filled = true
    this.radius = 1.0;
    System.out.println("Construced a Circle with Circle()");  // for debugging
  }
  public Circle(double radius) {
    super();                  //setting default color=red and filled = true
    this.radius = radius;
    System.out.println("Construced a Circle with Circle(radius)");  // for debugging
  }
  public Circle(double radius, String color, Boolean filled) {
    super(color, filled);   //setting given color and filled value
    this.radius = radius;
    System.out.println("Construced a Circle with Circle(radius, color)");  // for debugging
  }

  // public getters and setters for the protected variables
  public double getRadius() {
```

```
    return this.radius;
  }
  public void setRadius(double radius) {
    this.radius = radius;
  }

  /** Returns a self-descriptive String */
  @Override
  public String toString() {
    return  "Circle [“  +  super.toString() + ",  radius= " + this.radius +  "]";
  }

  /** The subclasses must implement the abstract methods: getArea() and getPerimeter() */
  /** Returns the area of this Circle */
  @Override
  public  double  getArea() {
    return   this.radius * this.radius * Math.PI;
  }

  /** Returns the perimeter of this Circle */
  @Override
  public  double  getPerimeter(){
        return  2  * radius  * Math.PI;
  }
}
```

The subclass: Rectangle

```
/**
 * The Rectangle class, subclass of Shape
 */

public class  Rectangle  extends  Shape {

   protected double length, width;   // Protected member variables

  /** Constructs overload – Default constructor - a Rectangle with default length and width */
  public Rectangle( ) {
    super( );
    this.length = 1.0;
    this.width =  1.0;
  }
  /** Constructs a Rectangle instance with the given length and width */
  public Rectangle(double width, double  length) {
    super( );
    this.length = length;
```

```
    this.width = width;
  }
  /** Constructs a Rectangle instance with the given length, width, color and filled */
  public Rectangle(double width, double  length, String color, Boolean filled) {
    super(color, filled);
    this.length = length;
    this.width = width;
  }

// public getters and setters for the protected variables
  public double getWidth() {
    return this.width;
  }
  public void setWidth(double width) {
    this.width = width;
  }
// public getters and setters for the protected variables
  public double getlength() {
    return this.length;
  }
  public void setLength(double length) {
    this.length = length;
  }

  /** Returns a self-descriptive string */
  @Override
  public String toString() {
    return   "Rectangle [" +  super.toString() + "width= " + this.width
                                        + " , length = " + this.length +  "]";
  }

/** The subclasses must implement the abstract methods: getArea() and getPerimeter() */
  /** Returns the area of the Rectangle */
  @Override
  public  double  getArea() {
    return   this.length*this.width;
  }
  /** Returns the perimeter of the Rectangle */
  @Override
  public  double  getPerimeter(){
        return  2  * ( this.length + this.width);
  }
}
```

The subclass of Rectangle: Square

```
/**
 * The Square class, subclass of Rectangle
 */
public class Square extends Rectangle {       // Save as "Square.java"

  // Constructor with default color, radius and height
  public Square() {
    super();      // call superclass no-arg constructor Shape() and Rectangle()
  }
  // Constructor with default radius, color but given height
  public Square(double side) {
    super(side, side);      // call superclass argument constructor Rectangle(width, length)
  }
  // Constructor with given values
  public Square(double side, String color, Boolean filled) {
    super(side, side, color, filled);
        // call superclass constructor Rectangle(width, length, color, filled)
  }

  // A public get/set methods for retrieving the side
  public double getSide() {
    return  super.getWidth();
  }
 public void setSide(double side) {
     super.setWidth(side);
  }
  public void setWidth(double side) {
    super.setWidth(side);
  }
  public void setLength(double side) {
    super.setLength (side);
  }

  /** Returns a self-descriptive string */
  @Override
  public String toString() {
    return  "Square [" + super.toString() + "]";
  }
}
```

Write **a test class** to test the following statements involving **polymorphism** and **explain the outputs.** Some statements may trigger compilation errors. **Explain the errors, if any**. A test driver class is as follows:

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

        Shape s1 = new Circle(5.5, "red", false);    // Upcast Circle to Shape
        System.out.println(s1);                      // which version?
        System.out.println(s1.getArea());            // which version?
        System.out.println(s1.getPerimeter());       // which version?
        System.out.println(s1.getColor());
        System.out.println(s1.isFilled());
        System.out.println(s1.getRadius());

        Circle c1 = (Circle)s1;                 // Downcast back to Circle
        System.out.println(c1);
        System.out.println(c1.getArea());
        System.out.println(c1.getPerimeter());
        System.out.println(c1.getColor());
        System.out.println(c1.isFilled());
        System.out.println(c1.getRadius());

        Shape s2 = new Shape();

        Shape s3 = new Rectangle(1.0, 2.0, "red", false);   // Upcast
        System.out.println(s3);
        System.out.println(s3.getArea());
        System.out.println(s3.getPerimeter());
        System.out.println(s3.getColor());
        System.out.println(s3.getLength());

        Rectangle r1 = (Rectangle)s3;                   // downcast
        System.out.println(r1);
        System.out.println(r1.getArea());
        System.out.println(r1.getColor());
        System.out.println(r1.getLength());

        Shape s4 = new Square(6.6);                      // Upcast
        System.out.println(s4);
        System.out.println(s4.getArea());
        System.out.println(s4.getColor());
        System.out.println(s4.getSide());

        // Take note that we downcast Shape s4 to Rectangle,
        //  which is a superclass of Square, instead of Square
        Rectangle r2 = (Rectangle)s4;
```

```
        System.out.println(r2);
        System.out.println(r2.getArea());
        System.out.println(r2.getColor());
        System.out.println(r2.getSide());
        System.out.println(r2.getLength());

        // Downcast Rectangle r2 to Square
        Square sq1 = (Square)r2;
        System.out.println(sq1);
        System.out.println(sq1.getArea());
        System.out.println(sq1.getColor());
        System.out.println(sq1.getSide());
        System.out.println(sq1.getLength());
 }
}
```

Check Your Understanding on Polymorphism

Q1. What is polymorphism in Java?
a) The ability of a variable to hold different data types
b) The concept of allowing methods to perform different tasks based on the object
c) The process of changing the state of an object
d) The ability to define multiple variables in a single declaration

Q2. What is method overloading in Java?
a) Redefining a method in a subclass
b) Changing the return type of a method
c) Creating multiple methods with the same name but different parameters in the same class
d) Extending a method from a superclass

Q3. What is method overriding in Java?
a) Providing a new implementation for an inherited method in a subclass
b) Creating a method with the same name and parameters in the same class
c) Changing the return type of a method in a subclass
d) Defining a method in an interface

Q4. Which keyword is used to achieve runtime polymorphism in Java?
a) static b) final c) super d) override

Q5. Can a static method be overridden in Java?
a) Yes b) No c) Only if it's in a superclass d) Only if it's in a subclass

Q6. Can interfaces in Java be used to achieve polymorphism?

a) Yes b) No c) Only if they are abstract d) Only if they extend a class

Q7. How does polymorphism benefit code reusability?

a) By using the same method for different purposes

b) By copying methods from one class to another

c) By allowing methods to run faster

d) By reducing the amount of memory used by the program

Q8. Concepts: Polymorphism

Question: Which lines where commented will the code successfully compile?

```
package com.markbdsouza.polymorphism;

interface Movable{   void move();  }

interface Eats{   void eat();  }

interface Vertebrae extends Movable, Eats{   }

class Dog implements Vertebrae {
   public void move() {
      System.out.println("Dog moves on 4 legs");
   }
   public void eat() {
      System.out.println("Dog eats food");
   }
}
public class  PolyTest1 {
   public static void main(String[] args) {
      // which lines when commented will the code successfully compile ?
      Movable a = new  Dog(); // line 17
      a.move(); // line 18
      a.eat(); // line 19
      Eats b = new Dog(); // line 20
      b.move();// line 21
      b.eat();// line 22
      Vertebrae c = new Dog(); // line 23
      c.move();// line 24
      c.eat();// line 25
   }
}
```

Q9. Concepts: Overloading + Overriding + Polymorphism

Question: What is the output of the below code?

```
package com.markbdsouza.polymorphism;

class Car {
   public int getSpeed() {  return  100;  }
}
class Audi extends Car {
   // overridden method as it already exists in the Car class
   public int getSpeed() {     return 500;   }
}
public class PolyTest1 {
   public void drive(Car car) {
      System.out.println("Using a Car and driving at " + car.getSpeed());
   }
   // overloaded version of drive method
   public void drive(Audi audi) {
      System.out.println("Using an Audi and driving at " + audi.getSpeed());
   }
   public static void main(String[] args) {
      // Create 3 objects
      Car car = new Car();
      Audi audi = new Audi();
      Car audiCar = new Audi();

      // Testing overloading and overriding with polymorphism
      PolyTest1    myTestClass = new   PolyTest1 ();
      myTestClass.drive(car);
      myTestClass.drive(audi);
      myTestClass.drive(audiCar);
   }
}
```

Q10. Write a Java program to create an interface Flyable with a method called fly_obj(). Create three classes Spacecraft, Airplane, and Helicopter that implement the Flyable interface. Implement the fly_obj() method for each of the three classes. Then write a test driver class that create object for each class and call the fly_obj() method that will display the type of aircraft flying, e.g. “Airplane is flying” for Airplane object. Use the following UML diagram to implement it.

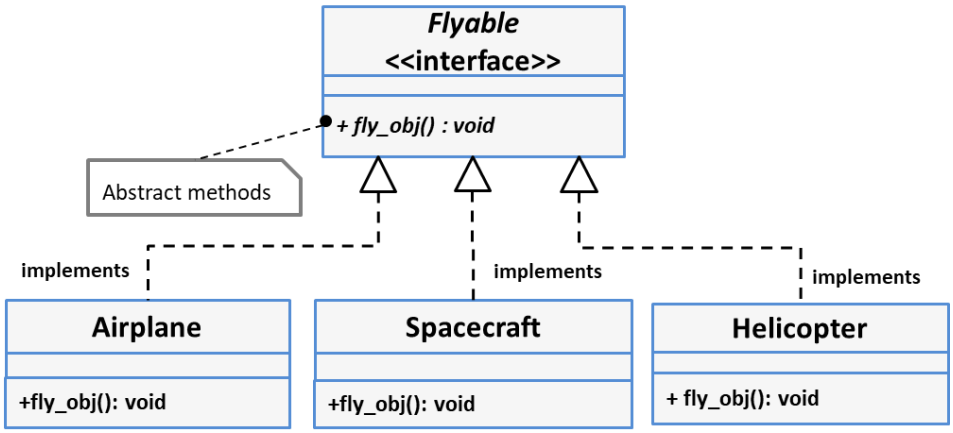


Figure Q10: Polymorphism and Interface

3.3 Composition Relationship Between Classes

Composition and aggregation represent **part/whole relationships among objects**. **Objects which contain other objects** are called **composite objects**. The composition/aggregation relationship is also known as **"has-a" relationship**.

- Aggregation/composition can occur in **a hierarchy of levels**. That is, an object contained in another object may itself contain **some other different object**.
- Aggregation/composition relationships **cannot be reflexive**. That is, an object cannot contain an **object of the same type as itself**.

Composition is a stricter form of aggregation, in which the **parts are existence-dependent on the whole**. This means that the **life of the parts cannot exist outside the whole.** In other words, the **lifeline of the whole and the part are identical**. When the whole is created, the parts are created and when the whole is destroyed, the parts are destroyed. **The Composition relationships among two classes A and B:**

- B is a permanent part of A
- A is made up of Bs
- A is a permanent collection of Bs

For example, consider "a Book is written by one Author", where the Book class has a composite relationship with the Author class who write the book. The Author object cannot exist without the Book object.

3.3.1 UML Notations for Composition Relationship

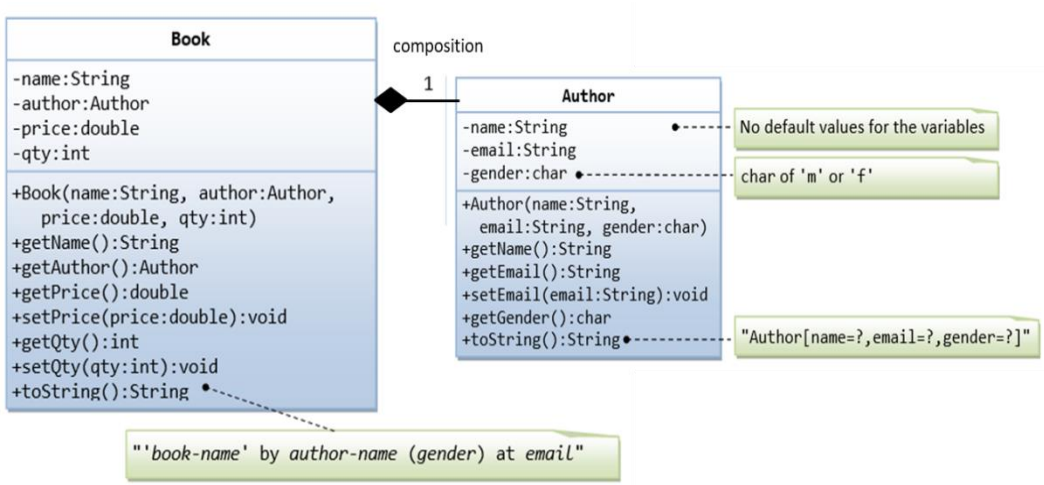


Figure 1.14: The composition relationship between Book and Author classes

Let's design a Book class, for the case that a book is written by one (and exactly one) author. The composition relation between Book class and the Author class shown in Figure 1.14. UML notation for composition relationship is represented as **a filled diamond drawn at the composite-end**.

In this book and author example, as soon as **a book object is created, the author object is also created** and as soon as **the book object is destroyed, the author object for that book is also destroyed**. That is, **the life of the component (author object) is the same as the aggregate (book object).**

3.3.2 Implementation Strategies for Composition Relationship

Composition relationship is formed when **a class has a reference to another class as an instance property**. With composition (aggregation), define a new class, which is composed of existing classes. In the composition relationships, a class that contains the reference to another class is the parent (owner) of that child class. The child class doesn't exist without the parent class.

The Book class contains the following members:

- Four private member variables: name (String), author (an *instance* of the Author class we have just created, assuming that each book has exactly one author), price (double), and qty (int).
- The public getters and setters: getName(), getAuthor(), getPrice(), setPrice(), getQty(), setQty().
- A toString() that returns "'book-name' by author-name (gender) at email". You could reuse the Author's toString() method, which returns "author-name (gender) at email".

Example: the Book class

```
/**
 * The Book class models a book with one (and only one) author.
 */
public class Book {
  // The private instance variables
  private String name;
  private Author author;
  private double price;
  private int qty;

  /** Constructs a Book instance with the given author */
  public Book(String name, Author author, double price, int qty) {
    this.name = name;
    this.author = author;
    this.price = price;
```

```
    this.qty = qty;
  }

  // Getters and Setters
  /** Returns the name of this book */
  public String getName() {
    return name;
  }
  /** Return the Author instance of this book */
  public Author getAuthor() {
    return author;    // return member author, which is an instance of the class Author
  }
  /** Returns the price */
  public double getPrice() {
    return price;
  }
  /** Sets the price */
  public void setPrice(double price) {
    this.price = price;
  }
  /** Returns the quantity */
  public int getQty() {
    return qty;
  }
  /** Sets the quantity */
  public void setQty(int qty) {
    this.qty = qty;
  }

  /** Returns a self-descriptive String */
  public String toString() {
    return "'" + name + "' by " + author;   // author.toString()
  }
}
```

A class called Author is designed as shown in the class diagram. It contains:

- Three private member variables: name (String), email (String), and `gender` (char of either `'m'` or `'f'` - you might also use a boolean variable called isMale having value of true or false).
- A constructor to initialize the name, email and gender with the given values. (There is no *default constructor*, as there is no default value for name, email and gender.)

- Public getters/setters: getName(), getEmail(), setEmail(), and getGender(). (There are no setters for name and gender, as these properties are not designed to be changed.)
- A toString() method that returns "name (gender) at email", e.g., "Tan Ah Teck (m) at ahTeck@somewhere.com".

Example: the Author class

```
/**
 * The Author class model a book's author.
 */
public class Author {
  // The private instance variables
  private String name;
  private String email;
  private char gender;   // 'm' or 'f'

  /** Constructs a Author instance with the given inputs */
  public Author(String name, String email, char gender) {
    this.name = name;
    this.email = email;
    this.gender = gender;
  }
  // The public getters and setters for the private instance variables.
  // No setter for name and gender as they are not designed to be changed.
  /** Returns the name */
  public String getName() {
    return name;
  }
  /** Returns the email */
  public String getEmail() {
    return email;
  }
  /** Sets the email */
  public void setEmail(String email) {
    this.email = email;
  }
   /** Returns the gender */
  public char getGender() {
    return gender;
  }
  /** Returns a self-descriptive String */
  public String toString() {
```

```
    return name + " (" + gender + ") at " + email;
  }
}
```

A Test Driver Program for the Book Class (TestBook.java)

```
/**
 * A test driver program for the Book class.
 */
public class TestBook {
  public static void main(String[] args) {
    // We need an Author instance to create a Book instance
    Author  smith = new Author("Smith  Will", "smith@somewhere.com", 'm');
    System.out.println(smith);  // Author's toString()
    // Smith  Will (m) at smith@somewhere.com

    // Test Book's constructor and toString()
    Book   dummyBook = new Book("Java for dummies", smith, 9.99, 99);
    System.out.println(dummyBook);  // Book's toString()
    //'Java for dummies' by Smith  Will (m) at smith @somewhere.com

    // Test Setters and Getters
    dummyBook.setPrice(8.88);
    dummyBook.setQty(88);
    System.out.println("name is: " + dummyBook.getName());
    //name is: Java for dummies
    System.out.println("price is: " + dummyBook.getPrice());
    //price is: 8.88
    System.out.println("qty is: " + dummyBook.getQty());
    //qty is: 88
    System.out.println("author is: " + dummyBook.getAuthor());  // invoke Author's toString()
    //author is: Smith  Will (m) at smith@somewhere.com
    System.out.println("author's name is: " + dummyBook.getAuthor().getName());
    //author's name is: Smith  Will
    System.out.println("author's email is: " + dummyBook.getAuthor().getEmail());
    //author's email is: smith@somewhere.com
    System.out.println("author's gender is: " + dummyBook.getAuthor().getGender());
    //author's gender is: m
    // Using an anonymous Author instance to create a Book instance
    Book moreDummyBook = new Book("Java for more dummies",
        new Author("Peter Lee", "peter@nowhere.com", 'm'), 19.99, 8);
    System.out.println(moreDummyBook);  // Book's toString()
    //'Java for more dummies' by Peter Lee (m) at peter@nowhere.com
  }
}
```

3.3.3 Summary of Composition Relationship:

- A class (parent) contains a reference to another class (child).
- The child class doesn't exist without the parent class.
- Deleting the parent class will also delete the child class.
- A class can also include a reference of the id property of another class instead of an instance of another class.

3.3.4 Exercise to Practice Composition Relationship

The composite relationship also has a **cardinality** that is a **one-to-one, one-to-many, or many-to-many relationship between classes**. In the above example, the Book and the Author class have a one-to-one relationship because each Book is written by only one author. Modify the **Book class to support one or more authors** by changing the instance variable authors to an **Author array**. For example, a composite relationship between Book and Author class for **a book can be written by one or more author**. The UML diagram is shown in Figure 1.15.

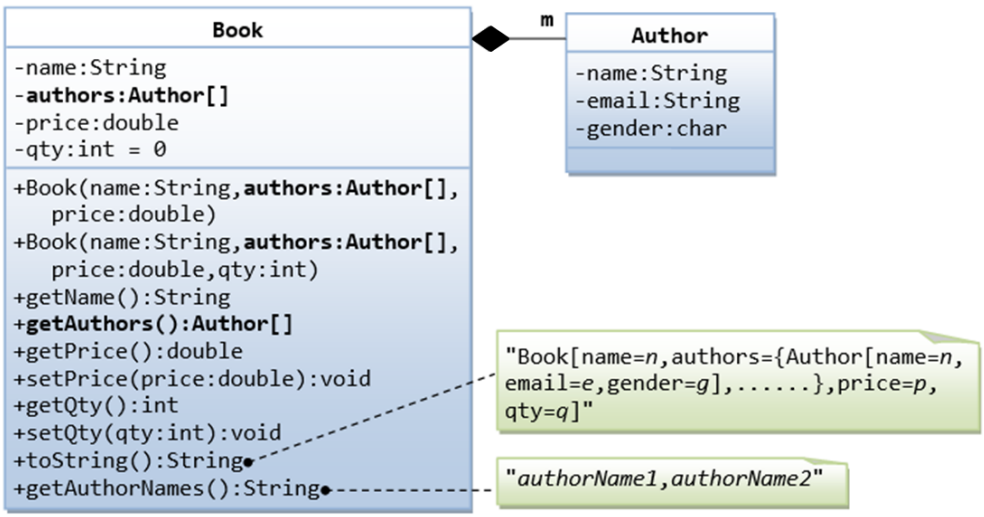


Figure 1.15: The composition relationship between Book and Author class for a book can be written by one or more author

Exercise: The Author and Book Classes - An Array of Objects as an Instance Variable

To implement **one to many** relationship between a book and number of authors, use an array of objects or collection instance as follows:

- The constructors take an array of Author (i.e., Author[]), instead of an Author instance. In this design, once a Book instance is constructor, you cannot add or remove author.

- The toString() method shall return "Book[name=?,authors={Author[name=?,email=?,gender=?],.....},price=?,qty=?]".
- To store one or more Authors for a Book, use Java Generic type Array List:

```
import java.util.List;
import java.util.ArrayList;

//Declare and allocate an array of Authors
List<Author>  authorLst = new ArrayList<>();
authorLst.add(new Author("John Smith", jsmith@somewhere.com", 'm'));
authorLst.add(new Author("Paw Lauw", plauw@somewhere.com", 'f'));

//allocate a Book instance with authors
Book  javaDummy  = new Book("Java for Dummy", authorLst, 19.99,99);
```

The Book class

```
/**
 * The Book class models a book with one or more authors.
 */

import java.util.List;
import java.util.ArrayList;

public class Book {
  // The private instance variables
  private  String  name;
  private  List<Author>   authorLst;   // Declare an array of Authors
  private  double  price;
  private  int  qty;

/** Constructs a Book instance with the given author with default price */
  public Book(String name, List<Author>  authors, double price) {
    this.name = name;
    this.authorLst = authors;
    this.price = price;
    this.qty = 0;
  }

/** Constructs a Book instance with the given author */
  public Book(String name, List<Author>  authors, double price, int qty) {
    this.name = name;
    this.authorLst = authors;
    this.price = price;
    this.qty = qty;
  }
```

```
  // Getters and Setters
  /** Returns the name of this book */
  public  String  getName() {
    return name;
  }

  /** Return the Author instance of this book */
  public  List<Author>  getAuthors() {
    return authors;    // return member author, which is an Array List of Author instances
  }
  /** Returns the price */
  public  double  getPrice() {
    return price;
  }
  /** Sets the price */
  public void setPrice(double price) {
    this.price = price;
  }
  /** Returns the quantity */
  public int getQty() {
    return qty;
  }
  /** Sets the quantity */
  public void setQty(int qty) {
    this.qty = qty;
  }
  /** Returns a self-descriptive String */
  public String toString() {
      String str  =  "Book[name= '" + name + "' by  authors = {";
      for (Author  auth : authorLst) {
          str = str  +  auth.toString();
      }
      str  =  str  +   "}, price = "  + price    +  " ,  qty = "  + qty + "]" ;
      return  str;   // book.toString()
  }
// Add authors to book
  public String getAuthorNames( ) {
     String str = "";
      for (Author  auth : authorLst) {
          str = str  +  auth.getName() +  " , " ;
      }
      return str;
   }
}
```

The Author class

```
/**
 * The Author class model a book's author.
 */
public class Author {
  // The private instance variables
  private String name;
  private String email;
  private char gender;   // 'm' or 'f'

  /** Constructs a Author instance with the given inputs */
  public Author(String name, String email, char gender) {
    this.name = name;
    this.email = email;
    this.gender = gender;
  }

  // The public getters and setters for the private instance variables.
  // No setter for name and gender as they are not designed to be changed.
  /** Returns the name */
  public String getName() {
    return name;
  }
  /** Returns the gender */
  public char getGender() {
    return gender;
  }
  /** Returns the email */
  public String getEmail() {
    return email;
  }
  /** Sets the email */
  public void setEmail(String email) {
    this.email = email;
  }

  /** Returns a self-descriptive String */
  public String toString() {
    return "Author[name= " +  name + ", gender= " + gender + " , email= " + email  + " ] ";
  }
}
```

A Test Driver Program for the Book Class (TestBook.java)

```
/**
 * A test driver program for the Book class.
 */
public class TestBook {
  public static void main(String[] args) {
```

```
    //Create Author instances for book
    List<Author>  authors = new  ArrayList<Author> ();

    // Add authors to array list - authors.add (new  Author(name, email, gender));
    authors.add (new  Author("Smith  Will", " smith@somewhere.com", "m"));
    authors.add(new  Author("Paw Lauw ", " plauw@somewhere.com", "f"));
    authors.add(new  Author("Marry  Brown", " mbrown@somewhere.com", "f"));

   // Test Book's constructor and toString()
    Book  javaDummy = new Book("Java for Dummies",  authors, 9.99, 99);

    System.out.println(javaDummy);  // Book's toString()
    // Book[name= 'Java for Dimmies' , authors = { Author[name= Smith  Will , gender= m ,
    // email=  smith@somewhere.com ], Author[name= Paw Lauw , gender= f , email=
    // plauw@somewhere.com ], Author[name= Marry Brown , gender= f , email=
    // mbrown@somewhere.com ] } , price = 9.99 , qty = 99 ]

    // Test Setters and Getters
    javaDummy.setPrice(8.88);
    javaDummy.setQty(88);
    System.out.println("name is: " + javaDummy.getName());
    //name is: Java for dummies
    System.out.println("price is: " + javaDummy.getPrice());
    //price is: 8.88
    System.out.println("qty is: " + javaDummy.getQty());
    //qty is: 88
   System.out.println("authors are: " + javaDummy.getAuthorNames());
    // authors are: Smith  Will,  Paw Lauw, Marry Brown ,
  }
}
```

Check Your Understanding on Composition Relationship Between Classes

Q1. Consider a Unicycle class and a Tire class. A Unicycle requires a Tire. In other words:

(a) The Tire HAS-A Unicycle (b) The Tire voids the Unicycle

(c) The Unicycle polymorphs the Tire (d) The Unicycle HAS-A Tire

Q2. Examine the following code. There are two classes: Page and Book. Which best describes their relationship?

(a) Book HAS-A Page (b) Page HAS-A Book

(b) Page inherits from Book (c) This code is invalid

Q3. Which one of the following relationships describes the OO design concept of "composition"?

(a) is-a (b) is-a-kind-of (c) has-a (d) composed-as

Q4. Composition in java is implemented by _____.

(a) a class contains instances of other classes to use their abilities
(b) a class references to objects to use link.
(c) a class inherits traits from a parent class
(d) a class defines the attributes of other classes

Q5. Composition is a design principle that represents a "whole-part" relationship between classes where one class represents a larger entity that is composed of smaller, reusable components and these components _____.

(a) can exist independently without the larger one
(b) cannot exist independently without the larger one
(c) can only exist with the main one
(d) none of the above

Q6. Consider the following java classes: Room and House classes. Is it the composition implementation in java? Explain Why? Draw a UML diagram for these two classes relationship.

```
class Room {

    private String name;

  public Room(String name) {
    this.name = name;
  }

  // Other room-related methods
}

class House {
  private List<Room> rooms;

  public House() {
    this.rooms = new ArrayList<>();
    rooms.add(new Room("Living Room"));
    rooms.add(new Room("Bedroom"));
    rooms.add(new Room("Kitchen"));
  }

  public List<Room>  getRooms() {
    return rooms;
  }
  // Other house-related methods
}
```

Q7. Let's consider a real-world example: a **Car** class that is made up of **Engine** and **Tyre** classes. In this example, the **Car** class contains an **Engine object** and an **array of Tyre objects**. If the Engine object or Tyre object gets destroyed then Car object cannot be completed, i.e., Car can not exist without these components. The relationship between the Car and its components is composition.

When you create a new **Car** instance, it immediately produces an instance of **Engine** and **four cases of Tyre**. This structure ensures the Car class has all its necessary components.This composition relationship emphasizes the concept that a car has an engine and has tires, emphasizing the importance of the link.

(a) Draw UML composition diagram of Car class made up of Engine and Tyre classes.

(b) Implement composition relationship of Car class made up of Engine and Tyre classes in java.

3.4 Aggregation Relationship between Classes

Aggregation is another category of "**has a**" **relationship** where **a class can contain other classes as properties** but those **classes can exist independently**.

Composition and Aggregation both are "has a" relationship but in the **composition relationship, related classes don't exist independently** whereas, in the **aggregation, related classes exist independently**.

For example, the Student has enrolled in number of courses and Student and Course have “has a” relationship. A Student class contains the Course class instance as a property to form the composition relationship. However, even if the Student object is deleted, the Course object will still exist. That is, the **related classes exist independently and both the classes can exist independently** and so it is called an aggregation relationship.

3.4.1 UML Notation for Aggregation Relationship between Classes

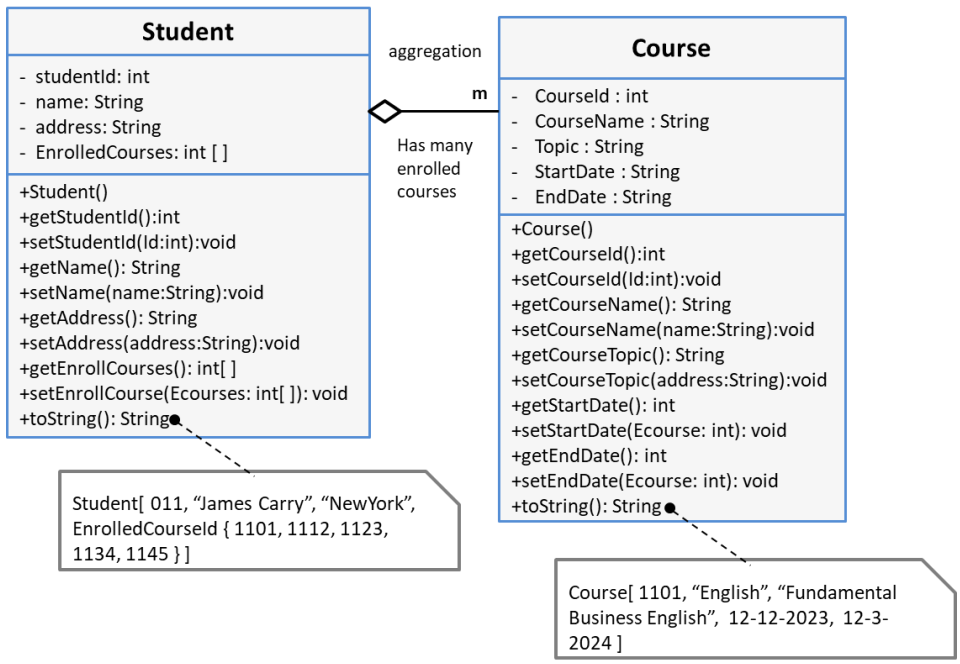


Figure 1.16: An aggregation relationship between Student and Course class

Let's design for a case that a Student has enrolled in number of courses. The aggregation relationship between a Student class and the Course class is shown in Figure 1.16. UML notation for aggregation relationship is represented as an **unfilled diamond drawn at the aggregate-end.**

In this example aggregation relationship, the Student object and the Course object will **exist independently.** The Student class can also contain CourseId property instead of Course instance.

3.4.2 Implementation Strategies for Aggregation Relationship

Aggregation or Composition relationship is formed when **a class has a reference to another class as an instance property**. With composition (aggregation), define a new class, which is composed of existing classes.

A class can also include the **id property of another class *instead of an instance*** to form the aggregation relationship. For example, the Student class can contain the CourseId property that points to the Course class. To keep the courses, we need to create CourseCatalog class. The code strategy is designed as follows:

Example: Aggregation Relationship – ID property instead of an Instance

The Student Class

```
/**
 * The Student class models a student has enrolled in number of courses.
 */
import java.util.List;
import java.util.ArrayList;

public class Student {
  // The private instance variables
  private  int  studentId;
  private  String  stdname;
  private  String  address;
  // A class can also include the id property of another class instead of an instance
  private  List<int>  enrollCourses;  // Declare an array of CourseId

/** Constructs a Student instance with the given values */
  public Student(int sId, String sname, String  address) {
    this.studentId  = sId;
    this.stdname = sname;
    this.address  =  address;
  }

  // Getters and Setters
  /** Returns the id of this student */
  public  int  getStudentId () {
    return this.studentId;
  }
```

```
  public void setStudentId (int Id) {
    this.studentId = Id;
  }
  /** Returns the name of this student */
  public String getName() {
    return this.stdname;
  }
  public void setName(String name) {
    this.stdname=name;
  }
  /** Returns the address of this student */
  public String getAddress() {
    return this.address;
  }
  public void setAddress(String address) {
    this.address = address;
  }
  /** Return the enrolled course List of this student*/
  public List<int> getEnrollCourses() {
    return this.enrollCourses;    // return an Array List of enrolled Course instances
  }
  public void setEnrollCourse(int enrollCourse) {
    this.enrollCourses.add(enrollCourse);
  }
  /** Returns a self-descriptive String */
  public String toString() {
      String str = "Student[ Id= '" + studentId + "' , " + stdname + " , " + address +
                                " , Enrolled Course ID = { ";
      for (int cid : enrollCourses) {
           str = str + cid + " , ";
      }
      str = str + " } ]" ;
      return str;  // Student.toString()
  }

}
```

The Course Class

```
/**
 * The Course class models course information.
 */

public class Course {
  // The private instance variables
```

```
  private int courseId;
  private String coursename;
  private String topic;
  private String startDate;
  private String endDate;

/** Constructs a Course instance with the given author */
  public Course (int CID, String cname, String topic, String sdate, String edate) {
    this.courseId = CID;
    this.coursename = cname;
    this.topic = topic; this.startDate = sdate; this.endDate = edate;
  }

  // Getters and Setters
  /** Returns the course id of this course */
  public int getCourseId() {
    return this.courseId;
  }
  /** Return the course name of this course */
  public String getCourseName() {
    return this.coursename;
  }
  /** Returns the Topic of Course */
  public String getTopic() {
    return this.topic;
  }
  /** Sets the Topic of Course */
  public void setTopic(String ctopic) {
    this.topic = ctopic;
  }
  /** Returns the start date of this course */
  public String getStartDate() {
    return this.startDate;
  }
  /** Sets the start date of this course */
  public void setStartDate(String sdate) {
    this.startDate = sdate;
  }
 /** Returns the end date of this course */
  public String getEndDate() {
    return this.endDate;
  }
  /** Sets the end date of this course */
  public void setEndDate(String edate) {
    this.endDate = edate;
```

```
  }
  /** Returns a self-descriptive String */
  public String toString() {
      String str = "Course[ CId= '" + courseId + " , " + coursename + " , " +
                                  topic + " , " + startDate + " , " + endDate + " ]" ;
      return str;  // course.toString()
  }
}
```

The CourseCatalog Class

```
/**
 * The Course Catalog class models list of courses information.
 */
import java.util.List;
import java.util.ArrayList;

public class CourseCatalog {
  // The private instance variables
  private List<Course>  courseLst;  // Declare an array of Authors

/** Constructs a Course Catalog  instance with the initial value */
  public CourseCatalog( ) {
    this.courseLst = new ArrayList<Course>();
  }
  // Add Course to Catalog
  public void addCourse(Course course) {
    courseLst.add(course);
  }
  //Remove Course from Catalog
  public void removeCourse(int pos) {
    courseLst.remove(pos);
  }
  // Search Course Id form  Catalog
  public Course searchCourse(int CId) {

      Course course = null;
      while (course : courseLst) {
          if (course.getCourseId() == Cid)
              return course;
      }
    return course;
  }
}
```

The Test driver Class

```
/**
 * A test driver program for the Book class.
 */
public class TestCourse {
  public static void main(String[] args) {

    //Create Course Catalog instances for courses
    CourseCatalog  ccatalog = new  CourseCatalog();

    // Add courses to catalog list -
   ccatalog.addCourse(new  Courses(1101, "English", "Fundamental Business English",
                                          "12-12-2023", "12-3-2024"));
   ccatalog.addCourse(new  Courses(1102, "Physic", "Fundamental Business English",
                                          "12-12-2023", "12-3-2024"));
   ccatalog.addCourse(new  Courses(1143, "Digital", "Digital Fundamental",
                                          "12-12-2023", "12-3-2024"));
   ccatalog.addCourse(new  Courses(1114, "ICS", "Information and Computing science",
                                          "12-12-2023", "12-3-2024"));
   ccatalog.addCourse(new  Courses(1201, "English", "Fundamental Business English",
                                          "1-5-2024", "12-8-2024"));
   ccatalog.addCourse(new  Courses(1212, "Programming", "Java Programming",
                                          "1-5-2024", "12-8-2024"));
   ccatalog.addCourse(new  Courses(1223, "Software Engineering",
                               "Introduction to Software Engineering", "1-5-2024", "12-8-2024"));
   ccatalog.addCourse(new  Courses(1214, "Data Structure",
                                "Data Structure", "1-5-2024", "12-8-2024"));

    // Student enrollment to courses
  public Student(int sId, String sname, String  address) {
   Student s1 = new Student(011, "Smith Will", "New York");
   s1.setEnrollCourse(1101);
   s1.setEnrollCourse(1102);
   s1.setEnrollCourse(1143);
   s1. setEnrollCourse(1114);

   Student s2 = new Student(012, "John Black", "Malbone");
   s2.setEnrollCourse(1101);
   s2.setEnrollCourse(1102);
   s2.setEnrollCourse(1143);
   s2. setEnrollCourse(1114);

   Student s3 = new Student(013, "Smith Will", "LogAnglis");
```

```
    s3.setEnrollCourse(1201);
    s3.setEnrollCourse(1212);
    s3.setEnrollCourse(1223);
    s3. setEnrollCourse(1214);

    //test getter and setter methods
    System.out.println(s1.toString());
    List<int>  enrollcourse =  s1.getEnrolledCourses();
    Course  ecourse;
    while( int  courseid : enrollcourse ){
          ecourse = ccatalog.searchCourse(courseid);
          if (ecourse != null)      System.out.println(ecourse.toString);
    }
   }
}
```

3.4.3 Summary of Aggregation Points:

- Aggregation is another type of composition ("has a" relation).
- A class (parent) contains a reference to another class (child) where **both classes can exist independently**.
- A class can also **include a reference of the id property of another class**.
- Aggregation is a special form of association where one class is composed of multiple instances of another class.
- In aggregation, the objects have a "has-a" relationship, where one object is a part or component of another object. This is different from regular association, where objects have a "knows-a" relationship.

Check Your Understanding on Aggregation Relationship between Classes

Q1. Aggregation represents a “has-a” relationship, where _____.

(a) one class contains objects of another class as part of its internal structure
(b) one object contains other objects as an essential part of the other class
(c) one object is related with other objects
(d) none of the above

Q2. Aggregation relationship is a weaker and more independent relationship of Composition.

(a) True (b) False

Q3. In Aggregation, one class acts as a container of another class objects (component). These components _____.

(a) can exist independently without the larger one

(b) cannot exist independently without the larger one
(c) can only exist with the main one
(d) none of the above

Q4. Consider the following classes. A Unicycle class and a Tire class. A Unicycle requires a Tire. In other words:

(a) The Tire HAS-A Unicycle
(b) The Tire voids the Unicycle
(c) The Unicycle polymorphs the Tire
(d) The Unicycle HAS-A Tire

Q5. Examine the following code. There are two classes: Page and Book. Which best describes their relationship?

```
Page page = new Page();
Book book = new Book(page);
```

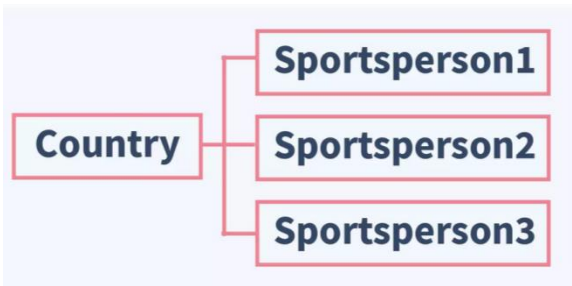
(e) Book HAS-A Page
(f) Page HAS-A Book
(g) Page inherits from Book
(h) This code is invalid

Q6. Consider the following java classes: Department and University classes. Is it the aggregation implementation in java? Explain Why? Draw a UML diagram for the relationship between Department and University classes.

```
class Department {
    private String name;

    public Department(String name) {
        this.name = name;
    }
    // Other department-related methods
}

class University {
    private String name;
    private List<Department> departments;

    public University(String name) {
        this.name = name;
        this.departments = new ArrayList<>();
```

```
    }

    public void addDepartment(Department department) {
        departments.add(department);
    }

    // Other university-related methods
}
```

Q7. Let's understand the aggregation in Java with the example of a Country class and a Sportsperson class.



- Country class is defined with a name and other attributes like size, population, capital, etc, and a list of all the Sportspersons that come from it.
- A Sportsperson class is defined with a name and other attributes like age, height, weight, etc.

A Country object has-a list of Sportsperson objects that are related to it. Note that a sportsperson object can exist with its own attributes and methods, alone without the association with the country object. Similarly, a country object can exist independently without any association to a sportsperson object. In, other words both Country and Sportsperson classes are independent although there is an association between them. Hence Aggregation is also known as a weak association.

(a) Draw UML Aggregation diagram of Country class and a Sportsperson class.
(b) Implement Aggregation relationship between Country class and a Sportsperson class for the case of a Country object has-a list of Sportsperson objects that are related to it in java.

3.5 Associations Relationship Between Classes

A more general relationship between the classes is known as an **association**. Associations describe the properties of the links that exist between objects when a system is running. A link from one object to another informs each object of the identity, or the location, of the other object. Links between objects enables the objects to send messages to each other, using the link as a kind of communication channel. For example, Figure 1.6 shows an association between Student and Module class that is modeling the fact that student takes many modules for a degree.

3.5.1 The UML diagram for Association

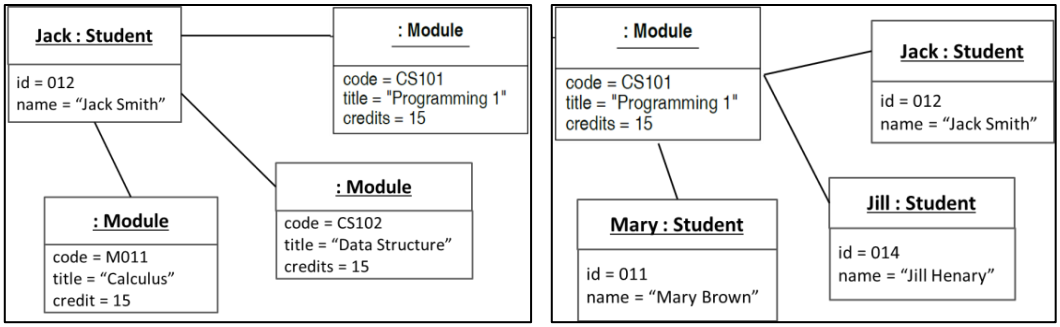
Associations are represented in UML as lines joining the related classes, shown in figure 1.17. The association is labeled with a **name**, 'Takes', which is shown near the middle of the association. The small triangle ('►') next to the association name indicates the **direction in which this sentence should be read**. For example, Figure 1.17 shows an association modeling the fact that **a student takes many modules**.

Figure 1.17: Association Relationship between classes

3.5.2 Links Relation between Objects

The existence of an association between two classes indicates that **instances of the classes can be linked at run-time**. A **link** is **a logical or physical connection between two or more objects**. Figure 1.18 shows a UML diagram example of a link specified by the association in Figure 1.17. Links can be labelled with the relevant association name, underlined to emphasize that the link is an instance of the association.

Figure 1.18 shows the facts that a student takes many modules, an instance of Student, Jack, links with three instances of Module: CS101, CS102, M011. Link defines the relationship between two or more objects and a link is considered as an instance of an association. This means **an association is a group of links that relates objects from the same classes.** Figure 1.18(a) and (b) show the state of the object diagrams that are legitimate instances of association in the corresponding class diagram given in Figure 1.17.



(a) A student may takes zero or more modules (b) A module may be taken zero or more students

Figure 1.18: The link between the instances of the association class

3.5.3 Multiplicity in Relations

The multiplicity specifies **how many objects an instance of the class at the other end of the association can be linked to**. Figure 1.19 shows the 'Takes' association with role names and multiplicity annotations ('*') is added at both end. This means that diagram specifies that a student takes more than zero or more modules and that a module can be taken by zero or more students.

Figure 1.19: Association Relationship between classes

The multiplicity is shown on each side of the association relation. It indicates how many instances of one class are associated with the other (or) specifies how many objects of the opposite class an object can be associated with. It is noted as an individual number or a range of the minimum and maximum values (syntax: number or min.. max), e.g. 1..5.

Multiplicity	Notation
zero	**0** or **0..0**
zero or one	**0..1**
many (zero or more)	**0..*** or *****
exactly one	**1**
one or more	**1..***

3.5.4 Navigability in Relations

The **navigability** in association relation is used to record the fact that certain links in the system could **only be traversed, or messages sent**, **in one direction**. This design decision can be shown on a class diagram by writing **a navigation arrow on the association** to show the required direction of traversal. It has been decided that an association only needs to be supported in one direction. It is like one way communication. **An association with no arrowheads**, such as the one in Figure 1.19, is normally assumed to be **navigable in both directions**.

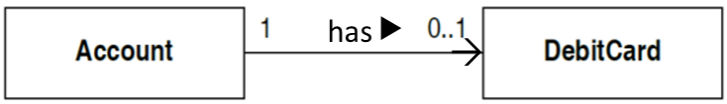


Figure 1.20: An association navigable in one direction only

For example, Figure 1.20 shows an association between Account and DebitCard class that is modeling the fact that every account can have a debit card issued for use with the account. It shows an association that is only to be implemented uni-directionally. This means that **an Account object holds a reference to at most one DebitCard object** and association would correspond to the requirement that **only one card was ever issued for a particular account.**

In other words, if the design decision was decided that there was **no need for DebitCard object to explicitly store a reference to the related Account object** because an Account object holds only one DebitCard object that is issued for, then, the association might be defined to be navigable in only one direction, as shown in Figure 1.20.

3.5.5 Implementation Strategies for Association Relation

Association relationship is referred to as "**uses a**" relationship where **a class uses another class to perform some operation.** In association, both classes can exist independently where nobody is the owner of another.

Association some refers as ***collaboration*** or ***delegation*** because association happens between the classes where **one class provides a service to another class** or the **class delegates some kinds of behaviors to another class**.

Association is typically **implemented with -**

- a pointer or
- reference instance variable or
- as a method argument.

Implementation strategies for mapping associations' relation between classes to the code can be done in following ways:

1. **Unidirectional Association**
   - Unidirectional Optional Associations
   - Unidirectional One-to-one associations / exactly one associations
   - Unidirectional One-to-many associations
2. **Bidirectional Association**
   - Bidirectional one-to-one and optional association
   - Bidirectional, one-to-many association
   - Bidirectional qualified association
3. **Association Classes**
4. **Qualified Association**
5. **Constraints**

Let's study each implementation strategies for class association with examples below.

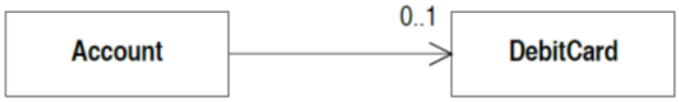
3.5.6 Unidirectional Association

An object might store another object in a field. This design decision decided that an association between classes only needs to be supported in one direction. The following sections discuss the cases where the multiplicity of the directed association is

- '**optional'**,
- '**exactly one'** and
- '**many'**.

(1) Unidirectional Optional Association

In optional association, a link between the participating objects may or may not exist. This type of association is implemented in one direction only. Figure 1.21 shows an association that is **only to be implemented uni-directionally**. For example, a bank offers a new facility that **every account** can have **a debit card** issued for use with the account, and it is assumed that many account holders will not immediately take up the chance to have such a card.



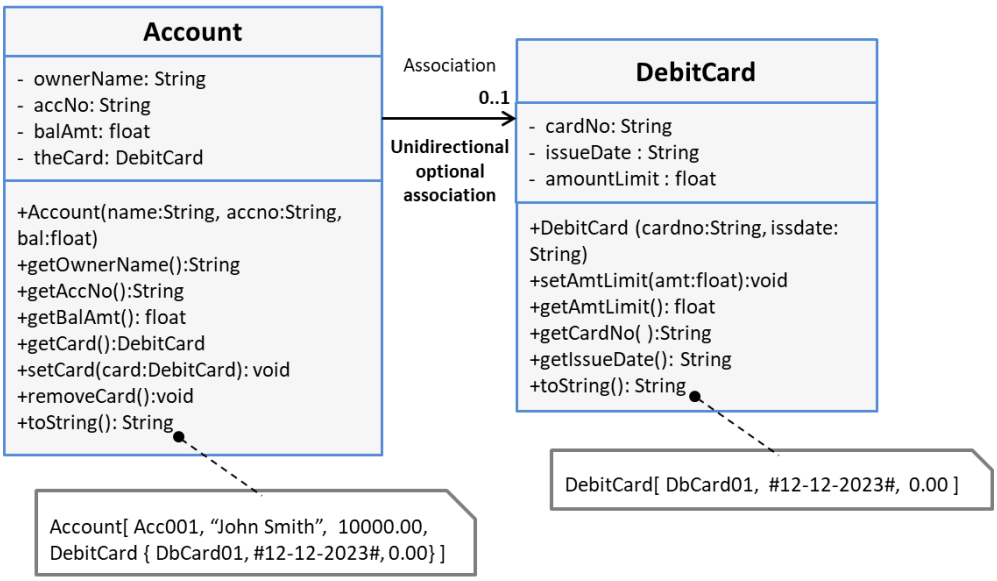


Figure 1.21: An optional association

This uni-directional optional association can be implemented using ***a simple reference variable***, as shown below code. In unidirectional association, you can traverse from an occurrence of one class to an occurrence of the associated class (as indicated by the direction of the arrow) but not in a reverse direction. That means Account class holds a link to the DebitCard Class but not in a reverse direction.

This allows an **Account** object to ***hold a reference to at most one DebitCard object***. Cases where an **account** is not linked to a card are modeled by allowing the reference variable to ***hold a null reference***. This implementation therefore provides exactly the **multiplicity of zero or one** requirements specified by the association. The code design for this condition is as follow:

```
public class Account{

   private  DebitCard  theCard ;
   public DebitCard  getCard() {
            return theCard;
   }
   public void  setCard (DebitCard card) {
   // an account object to hold a reference to
   //at most one debit card object
            theCard = card;        }

   public void removeCard() {
   // multiplicity of zero or one variable to
   // hold a null reference.
```

```
public class DebitCard {

   private  String cardno;
   private  String  issueDate;
   private  float  amountLimit;

   DebitCard(String cardno, String issueDate)
   {
        this. cardno =  cardno;
        this.issueDate  = issueDate;
   }

   public void setAmtLimit(float  amt){
       this.amountLimit = amt;
   }
```

```
            theCard = null;
    }
    ....
    private String ownerName;
    private String accNo;
    private  float balAmt;
    Account(String name, String accno, float
    bal){
          this.ownerName = name;
          this.accNo = accno;
          this.balAmt = bal;
          this.theCard = null;
    }
    ... getter() & setter() for each attribute
}
```

```
  public  float getAmtLimit() {
    return this.amountLimit;
  }
  ...
  public String getCardNo(){
    return this.cardno;
  }
  public String getIssueDate(){
    return this.issueDate;
  }
  ...
}
```

In the above example, only the `Account` class holds a reference variable of `DebitCard` class and also uses the `DebitCard` class as a parameter of methods in setCard() method. This allows when an `Account` object is created, it ***holds a reference to at most one*** `DebitCard` ***object***. There is no `Account` class reference is stored in the `DebitCard` class. So the system could **only be traversed, or messages sent**, **in one direction from** `Account` **object to the** `DebitCard` **object**. The implementation design for the account class, debit card class and test driver class for unidirectional optional association is as follow:

The Account Class

```
/**
 * The Account class holds a reference variable (at most one) the DebitCard object of DebitCard
 *  class.
 */

  public class Account{

    private String   ownerName;
    private String   accNo;
    private  float   balAmt;
    private   DebitCard   theCard ;
    Account(String name, String accno, float  bal){
          this.ownerName = name;
          this.accNo = accno;
          this.balAmt = bal;
          this.theCard = null;
```

```
    }
    public String getOwnerName(){
       return ownerName;
    }
    public String getAccNo(){
       return accNo;
    }
    public float getBalAmt(){
       return balAmt;
    }
    public DebitCard getCard() {
            return theCard;
    }
    public void setCard (DebitCard card) {
    // an account object to hold a reference to at most one debit card object
            theCard = card;
    }
    public void removeCard() {
    // multiplicity of zero or one variable to hold a null reference.
            theCard = null;
    }
    public String toString(){
        String str = "Account[ accNo + " , " + ownerName + " , " + balAmt + " , " +
                    " , DebitCard { " + theCard.getCardNo() + " , " +
        theCard.getIssueDate() +
                    " , " + theCard.getAmtLimit() + "} ] " ;
        return str;
    }
}
```

The DebitCard Class

```
/**
 * The DebitCard class holds no Account class reference. The messages sent, in one direction
 *  from Account object to the DebitCard object.
 */

public class DebitCard {

   private String cardno;
   private String issueDate;
   private float amountLimit;

   DebitCard(String cardno, String issueDate) {
```

```
        this. cardno =  cardno;
        this.issueDate  = issueDate;
    }
    public void setAmtLimit(float  amt){
        this.amountLimit = amt;
    }
    public  float getAmtLimit() {
      return this.amountLimit;
    }
    public String getCardNo(){
      return this.cardno;
    }
    public String getIssueDate(){
      return this.issueDate;
    }
    public String  toString(){
      return  "DebitCard [ "  +  cardno  +  " , "  +  issueDate  +   " ,  "   +  amountLimit + " ] " ;
    }
}
```

The Test Deriver Class

```
public class TestAccount {

    public static void main(String[] args) {

          Account acc1 = new Account("John Smith", "Acc001", 10000);
          Account acc2 = new Account("Henary Joe", "Acc002", 30000);
          Account acc3 = new Account("Mary Brown", "Acc003", 50000);

          DebitCard  DbCard1 = new DebitCard("DbCard01", "#12/12/2001#");
          DebitCard  DbCard2 = new DebitCard("DbCard02", "#1/1/2020#");
          DebitCard  DbCard3 = new DebitCard("DbCard03", "#10/5/2022#");

          acc1.setCard(DbCard1);   //debit card 1 is issued for acc1.
          acc2.setCard(DbCard2);  // debit card 2 is issued for acc2.

          //Display each account with its debit card
          System.out.println("The account owner: " + acc1.getName() +
                                  "has a debit card: " + acc1.getCard().getCardNo());

         // The account owner: John Smith has a debit card: DbCard01
          System.out.println("The account owner: " + acc2.getName() +
                                  " has a debit card: " + acc2.getCard().getCardNo());
        // The account owner: Henary Joe has a debit card: DbCard02

          //following statement  violate the multiplicity 0 or one
          //at most one DebitCard object is issued for one account.
```

```
        acc1.setCard(DbCard3);  // debit card 3 is also issued for acc1.

        System.out.println("The account owner: " + acc1.getName() +
                                " has a debit card: " + acc1.getCard().getCardNo());
       // The account owner: John Smith has a debit card: DbCard03

        }
}
```

The execution output of the `TestAccount.java` is shown below, as two account objects: `acc1` and `acc2`, each account has a debit card object: `DbCard1` and `DbCard2`. In the `TestAccount` class, the account object, `acc1`, is assigned to two debit card objects: `DbCard1` and `DbCard3`, that is the violation of multiplicity zero or one. **Consider how to prevent issuing two debit cards to one account.**

The Output of execution of the `TestAccount.java`

```
The account owner: John Smith has a debit card: DbCard01
The account owner: Henary Joe has a debit card: DbCard02
The account owner: John Smith has a debit card: DbCard03
```

1-1 Immutable Associations

In object-oriented programming, **an immutable object is unchangeable object**, and **its state cannot be modified after it is created**, whereas **mutable objects are changeable objects** which **can be modified after their creation**. Immutable objects are inherently **thread-safe** and offer **higher security** than mutable objects. Associations are immutable data structures. This means that they carry no state and a copy of an Association is another completely independent Association.

Consider the association between Account and DebitCard where the **requirement is that only one debitcard was ever issued for a particular account.** This asspocation is called **Immutable associations**, i.e., **a link to one object cannot subsequently be replaced by a link to a separate object**. If the association between Accounts and Debit Cards was intended to be immutable, an operation to ***add a card to an account and only allows a card to be added if no card is already held***. Once allocated, a card cannot be changed or even removed. See the code below for immutable association.

```
public class Account {

    private  DebitCard  theCard;
    public Account(DebitCard dcard)  {
```

```
            theCard = dcard;
    }
    public DebitCard getCard() {
        return theCard ;
    }
    public void setCard(DebitCard card) {
        if (theCard != null) {  // Immutable associations
            // throw ImmutableAssociationError
        }
        theCard = card ;
    }
    ....
}
```

1-2 Mutable Associations

As described above, the mutable objects are changeable objects which can be modified after their creation. Consider the association between Account and DebitCard, if the requirement allows **different cards to be linked to an account at different times** during its lifetime, associations with this property are sometimes called **mutable associations** where **multiplicity become one or more** between `Account` and `Debit Card class`.

(2) Unidirectional One-to-One Associations

The one-to-one association is a simplest form of association to implement between two classes. In one-to-one association, one occurrence of a class is related to exactly one occurrence of the associated class. This example of an **immutable association** demonstrates that in general only some of **the properties of associations** can be **implemented directly by providing suitable declarations of data members in the relevant classes.**

Consider the association shown in Figure 1.22. This association describes a situation where **bank accounts must have a guarantor who will underwrite any debts incurred by the account holder**. It may frequently be necessary to find the details of the guarantor of an account, but in general it will not be necessary to find the account or accounts that an individual guarantor is responsible for. It has **therefore been decided to implement** the **association only in the direction from account to guarantor**.

The previous example, in figure 1.21, showed that a variable holding a reference has a natural multiplicity of zero or one, because it can hold a null reference. **If a multiplicity of exactly one is required**, additional code must be written to **check at run-time for the presence of a null reference.** In the following implementation of the `Account` class, ***the constructor throws an exception if a null reference to a guarantor object is provided***, and no operation is defined in the interface to update the reference held.

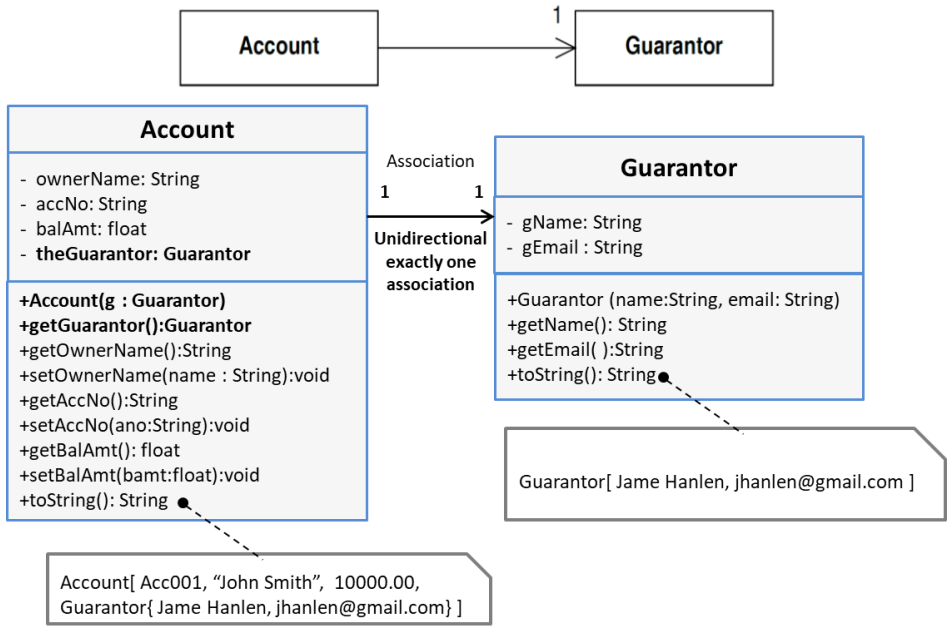


Figure 1.22: A one-to-one association

```
public class Account {

    private Guarantor theGuarantor;
    public Account(Guarantor g) {
      if ( g == null ) {
        //Immutable associations:
        // multiplicity of exactly one
              // throw NullLinkError
      }
      theGuarantor = g ;
    }

    public Guarantor getGuarantor() {
        return theGuarantor ;
    }
    public void setGuarantor(Guarantor g) {
      if (g == null) {
          // throw NullLinkError
      }
      theGuarantor = g ;
    }
}
```

```
public class Guarantor {

    private String name;
    private String email;

    public Guarantor(String name, String
    email ) {
      this.name = name;
      this.email = email;
    }

    public String getName() {
      return this.name;
    }
    public String getEmail(){
      return this.email;
    }
}
```

This code implements the **association between account and guarantor objects as an immutable association**. If the association was **mutable**, so that the ***guarantor of an account could be changed***, a suitable function could be added to this class provided that, like the constructor, it checked that the new guarantor reference was non-null.

The Account Class

```
/**
 * The Account class holds a reference variable (exactly one) the Guarantor object.
 */
  public class Account{

     private String   ownerName;
     private String   accNo;
     private  float   balAmt;
     private  Guarantor   theGuarantor;
     public Account(Guarantor   g){
          if ( g == null ) {
            //Immutable associations:  multiplicity of exactly one
                    // throw NullLinkError
          }
          theGuarantor = g ;

     }
     public Guarantor getGuarantor() {
          return  theGuarantor ;
     }
     public void setGuarantor(Guarantor g) {
       if (g == null) {
            // throw NullLinkError
       }
       theGuarantor = g ;
     }
     public  String  getOwnerName(){
        return  ownerName;
     }
     public  void  setOwnerName(String  name){
        this.ownerName = name;
     }
     public  String  getAccNo(){
        return  accNo;
     }
     public  void  setAccNo(String   acc){
        this. accNo =  acc;
```

```
    }
    public float getBalAmt(){
       return balAmt;
    }
    public void setBalAmt(float  bal){
       this.balAmt = bal;
    }
    public String toString(){
        String str = "Account[ accNo + " , " + ownerName + " , " + balAmt + " , " +
                     theGuarantor.toString()  +  " ] " ;
        return str;
    }
}
```

The Guarantor Class

```
/**
 * The Guarantor class holds no Account class reference. The messages sent, in one direction
 * from Account object to the Guarantor object.
 */

public class Guarantor {

   private String name;
   private String email;

   public Guarantor(String name, String email ) {
        this.name = name;
        this.address = email;
   }

   public String getName() {
        return  this.name;
   }
   public String getEmail(){
      return this.email;
   }
   public String toString(){
      return " Guarantor [ " + name + " , " + email +  " ] " ;
   }
}
```

The Test Driver Class

```
public class TestGuarantor {

  public static void main(String[] args) {
        //Create account objects
        Account acc1 = new Account (new Guarantor("Jame Hanlen", "jhanlen@gmail.com"));
        Account acc2 = new Account (new Guarantor("Horgan Will", "mwill@gmail.com"));
        Account acc3 = new Account (null);    // this will throw null exception error

        acc1.setOwnerName("John Smith");  acc1.setAccNo( "Acc001");
        acc1.setBalAmt(10000.00);
        acc2.setOwnerName("Henary Joe"); acc2.setAccNo( "Acc002");
        acc2.setBalAmt( 30000.00);

        //get the list of accounts that manager is responsible for
        System.out.println(acc1.toString());
        System.out.println(acc2.toString());
  }
}
```

(3) Unidirectional One-to-Many Association

This type of association exists in those cases where **one instance of a class is related to many instances of associated class**. Figure 1.23 shows an **association with multiplicity 'many'** for which **a unidirectional implementation** is required. Each manager within the bank is responsible for looking after a number of accounts, but the model assumes that ***it is never necessary to find out directly who the manager of any particular account is***. The new feature of this association is that a manager object could be linked not just to one, but potentially too many account objects.

In order to implement this association the **manager object must maintain multiple pointers, one to each linked account**, and hence **some suitable data structure must be used to store all the pointers**. In addition, it is likely that the interface of the manager class will provide operations to maintain the collection of pointers: for example, to add or remove details of particular accounts.

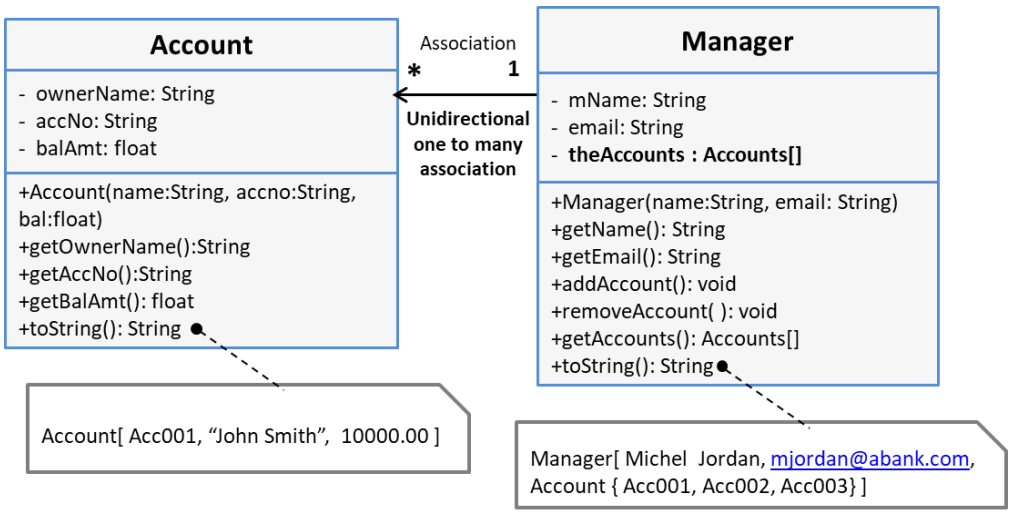


Figure 1.23: An association with multiplicity 'many'

The simplest and most reliable way to implement such an association is to make use of a suitable **container class from a class library**. For simple implementations in **Java**, the **Vector** class is a natural choice for this purpose. A skeleton implementation of the manager class using vectors is given below. The class declares a vector of accounts as a private data member, and the functions to add and remove accounts from this collection simply call the corresponding functions defined in the interface of the vector class.

The Manager Class

```
/**
 * The Manager class holds numbers of Account objects. The messages sent, in one direction
 * from Manager object to the Account objects.
 */
  public class Manager {

     private String mname;
     private String email ;
     // multiplicity 'many' maintain multiple pointers one to each linked account
     private List<Account>   theAccounts ;
     public Manager(String name, String email){
              this.mname = name;
              this.email = email;
              this.theAccounts = new ArrayList<Account>();
     }
     public String getName(){
              return this.maname;
     }
```

```
    public String getEmail(){
            return this.email;
    }
    public void addAccount(Account acc) {
             theAccounts.addElement(acc) ;
    }
    public void removeAccount(Account acc) {
             theAccounts.removeElement(acc) ;
    }

    public List<Account>  getAccounts(){
                return  theAccounts
    }

    public  void  toString(){
          System.out.print("Manager [ " +  mname  +  " , "  + email   +   " Accounts{  "  );
          for(Account   acc : theAccounts){
              System.out.prin (acc.getAccNo()  + " , " ) ;
          }
          System.out.println( " } ] ");
    }
}
```

Part of the semantics of a class diagram such as Figure 1.23 is that there can be **at most one link between a manager and any particular account**. In this implementation, however, there is no reason why many pointers to the same account object could not be stored in the vector held by the manager. This is a further example of the inability of programming languages to capture in declarations every constraint expressible in UML's class diagrams. **A correct implementation of the `addAccount` function should check that the account being added is not already linked to the manager object**.

The Account Class

```
/**
 * The Account class holds no reference Manager object. The message sent only
 *  from manager object to account objects.
 */
public class Account {

    private String ownerName;
    private String accNo;
    private float  balAmt;
    //Constrators
```

```
    Account(String name, String accno, float  bal){
          this.ownerName = name;
          this.accNo = accno;
          this.balAmt = bal;
    }
    public String getOwnerName(){
           return  this.name;
    }
    public String getAccNo(){
           return   this.accNo;
    }
    public  float  getBalAmt(){
          return   this.balAmt;
    }
    public  void  toString(){
          System.out.prinln("Account [ " +  this.accNo  + " , "  +  this.name  +  " , "  + this.balAmt
                    +  " ] "  );
    }
 }
```

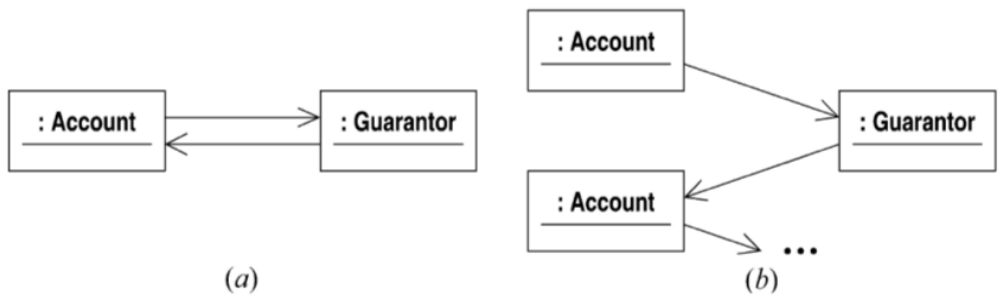
The Test deriver Class

```
 public class TestManager {

   public static void main(String[] args) {
        //Create account objects
        Account acc1 = new Account("John Smith", "Acc001", 10000);
        Account acc2 = new Account("Henary Joe", "Acc002", 30000);
        Account acc3 = new Account("Mary Brown", "Acc003", 50000);

        //Create manager objects
        Manager manager1 = new Manager("Michel Jordan", "mjordan@abank.com");

        //Assign accounts to manager
        manager1.addAccount(acc1);
        manager1.addAccount(acc2);
        manager1.addAccount(acc3);

        //get the list of accounts that manager is responsible for
        System.out.println(manager1.toString());
   }
}
```

3.5.7 Bidirectional Association

Bidirectional Association is a type of association where **one object is related with other objects and the reverse is also true**. It is like **two way communication**. If a bidirectional implementation of an association is required, **each link could be implemented by a pair of references**. This implementation requires the data members necessary to **store the bidirectional links** and requires **suitable fields must be declared in both classes** participating in the association.

The extra complexity involved in dealing with bidirectional implementations arises from the need to ensure that **at run-time the two pointers implementing a link are kept consistent**. This property is often referred to as **referential integrity**. Figure 1.24(a) shows the desired situation, where **a pair of 'equal and opposite' pointers implement a single link**. By contrast, Figure 1.24(b) shows the **violation of referential integrity**.



(a) Correct referential integrity (b) Incorrect referential integrity

Figure 1.24: Referential integrity in bidirectional association

Consider the association between accounts and guarantors, the required property can be stated informally as '**the guarantor of an account must guarantee that same account'**. Figure 1.24(b) violates this: the top account object holds a reference to a guarantor object, which in turn holds a reference to a completely different account. These two references cannot be understood as being an implementation of a single link. It should be clear from this example that **referential integrity cannot be ensured by simply giving appropriate definitions of data members in the relevant classes**.

(1) Bidirectional One-to-One and Optional Associations

Assume that the association in Figure 1.25 is **immutable in the debit card to account** direction, or in other words that, **once a debit card is linked to an account, it must stay linked to that account until the end of its lifetime**. **An account**, on the other hand, **can have different cards associated with it at different times**, to cater for situations where the account holder loses a card, for example.

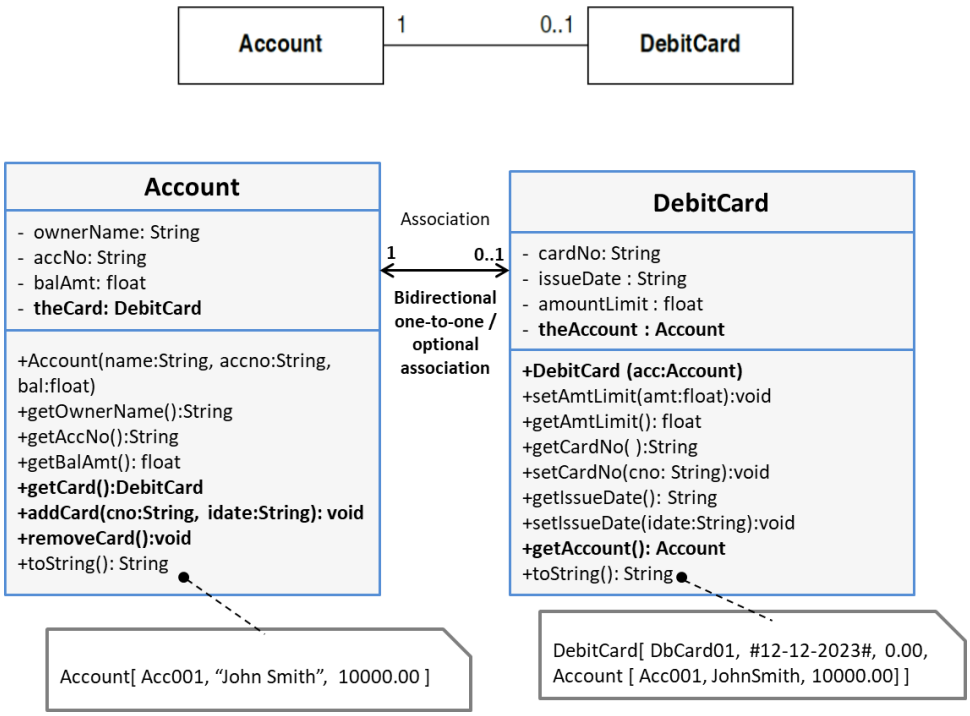


Figure 1.25: A bidirectional association

Example 1: A combination of a mutable and an immutable association

This association can be thought of as **a combination of a mutable and optional association in the left-to-right direction** with **an immutable association in the other**. A simple approach to its implementation would simply combine the implementations of those associations as shown below. For simplicity, the bodies of the methods in the classes are omitted. The complete example is described later in this section.

```
public class Account
{
  private DebitCard theCard ;
  public DebitCard getCard()
  {     return theCard;     }
  public void setCard(DebitCard card)
  {   theCard = card; }
  public void removeCard()
  {   theCard = null; }
  ... }
```

```
public class DebitCard
{
  private Account theAccount ;
  public DebitCard(Account a)
  { theAccount = a; }
  public Account getAccount()
  { return theAccount; }
  ...
}
```

This implementation certainly provides the data members necessary to store the bidirectional links, but the methods maintain the two directions of the link independently. For example, to **create a link between a new debit card and an account two separate operations are required, first to create the card and second to link it to the account**. The link from card to account is created when the card itself is created. Code to implement this might be as follows.

```
Account acc1 = new Account() ;
DebitCard card1 = new DebitCard(acc1) ;
acc1.setCard(card1) ;
```

To ensure that referential integrity is maintained, it is necessary to ensure that these two operations are always performed together. However, as two separate statements are needed, there is a real possibility that one may be omitted, or an erroneous parameter supplied, leading to an inconsistent data structure, as in the following example, in which **the debit card is initialized with the wrong account**.

```
Account  acc1 = new Account();
Account  acc2 = new Account() ;
DebitCard card1 = new DebitCard(acc2) ;
acc1.setCard(card1) ;
```

A better solution is explicitly to give the responsibility for maintaining the association to one of the classes involved. The choice of which class to give the maintenance responsibility to often arises naturally out of other aspects of the overall design. In the current case, it is likely that there would be an operation on the **account class to create a new debit card for the account** and this would provide a strong argument for making the **account class responsible for maintaining the association**. If this was the case, the classes could be defined as follows.

```
public class Account
{
   private DebitCard theCard ;
   public  DebitCard  getCard() {
      return theCard;
    }
   public void addCard(String cno, String idate) {
      theCard = new DebitCard(this);
      theCard.setCardNo(cno);
      theCard.setIssueDate(idate);
      thCard.setAmtLimit(10000.00);
   }
```

```
public class DebitCard
{
   private Account theAccount ;
   public DebitCard(Account a) {
        theAccount = a;
   }
   public Account getAccount() {
     return theAccount;
   }
....
}
```

```
   public void removeCard() {
     theCard = null; }
}
```

This example is particularly simple because the association was declared to be **immutable in one direction**. In general, if both directions of an association are mutable, a wide range of situations can arise in which links can be altered, and a correct implementation must ensure that these are all correctly handled.

The Account Class

```
/**
 * The Account class holds a reference variable (at most one) the DebitCard object of DebitCard
 *  class. This association is a combination of a mutable and optional association
 *  in the left-to-right direction with an immutable association in the DebitCard.
 */

   public class Account{

      private String   ownerName;
      private String   accNo;
      private  float   balAmt;
      private   DebitCard   theCard ;
      Account(String name, String accno, float bal){
            this.ownerName = name;
            this.accNo = accno;
            this.balAmt = bal;
            this.theCard = null;
      }
      public  String  getOwnerName(){
         return  ownerName;
      }
      public  String  getAccNo(){
         return  accNo;
      }
      public  float  getBalAmt(){
         return  balAmt;
      }
      public DebitCard  getCard() {
                  return  theCard;
      }
      public void addCard(String cno, String idate) {
            // an account object to hold a reference to  at most one debit card object
            theCard = new DebitCard(this);
```

```
            theCard.setCardNo(cno);
            theCard.setIssueDate(idate);
            theCard.setAmtLimit(10000.00);  // default amount is set
      }
      public void removeCard() {
      // multiplicity of zero or one variable to hold a null reference.
                  theCard = null;
      }
      public  String  toString(){
            return  "Account[ " + accNo + " ," + ownerName + " ," +  balAmt   + " ] " ;
      }
}
```

The DebitCard Class

```
/**
 * The DebitCard class holds exactly one Account object reference.
 * The DebitCard object is created from the Account object.
 *  This association  is  a combination of a mutable and optional association in the left-to-right
 *  direction with an immutable association in the DebitCard class.
 */
public class DebitCard {

   private  String cardno;
   private  String  issueDate;
   private  float  amountLimit;
   private  Account  theAccount ;

   public DebitCard(Account  a) {
        this.theAccount = a;
   }
   public  Account  getAccount() {
     return this. theAccount;
   }
   public  void  setCardNo(String  cno){
       this.cardno = cno;
   }
   public String getCardNo(){
     return this.cardno;
   }
   public void setAmtLimit(float  amt){
        this.amountLimit = amt;
   }
   public  float   getAmtLimit() {
     return this.amountLimit;
```

```
  }
  public String getIssueDate(){
    return this.issueDate;
  }
  public  void  setIssueDate(String  idate){
    this.issueDate  = idate;
  }
  public  String  toString(){
     return  "DebitCard [ "  +  cardno  +  " , "  +  issueDate  +   " ,  "   +  amountLimit  +
               theAccount.toString()   +  "  ] " ;
  }
}
```

The Test Deriver Class

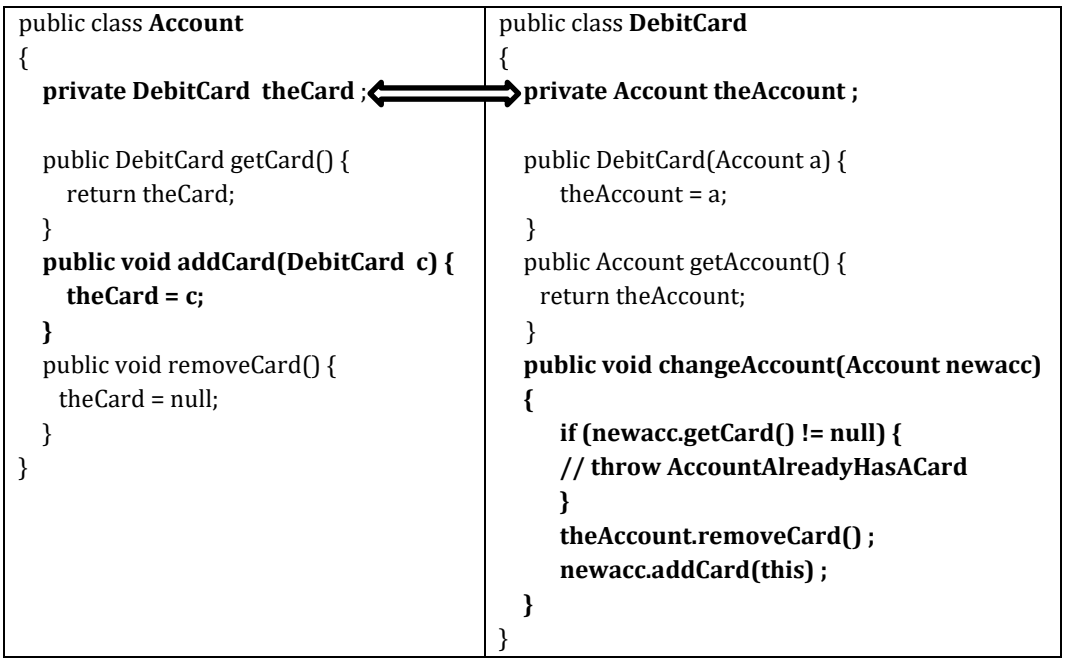
```
 public class TestDebitCard {
  //testing Account and Debitcard association with immutable in one direction
  public static void main(String[] args) {
        //Create account objects
        Account acc1 = new Account("John Smith", "Acc001", 10000);
        Account acc2 = new Account("Henary Joe", "Acc002", 30000);
        Account acc3 = new Account("Mary Brown", "Acc003", 50000);

        //Create DebitCard from Account objects
        acc1.addCard("DbCard01", "#12/12/2001#");
        acc2.addCard("DbCard02", "#1/1/2020#");
        acc3.addCard("DbCard03", "#10/5/2022#");

        //get the debit card that account is responsible for
        System.out.println(acc1.getCard().toString());
        System.out.println(acc2.getCard().toString());
        System.out.println(acc3.getCard().toString());

// DebitCard[ DbCard01,  #12/12/2001#,  10000.00, Account {Acc001, JohnSmith, 10000.00} ]
// DebitCard[ DbCard02,  #1/1/2020#,  10000.00, Account {Acc002, JohnSmith, 30000.00} ]
// DebitCard[ DbCard03,  #10/5/2022#,  10000.00, Account {Acc003, JohnSmith, 50000.00} ]
  }
}
```

Example 2: Mutable Association in Both Directions

For example, suppose that a customer can hold many accounts but only one debit card, and has the facility to nominate which account is debited when the card is used. **The association between accounts and debit cards will now be mutable in both directions**

and it will be reasonable for the card class to provide an operation to change the account that the card is associated with.

The implementation sketched below follows the strategy given above of allocating the responsibility of manipulating references exclusively to the account class. As explained above, this makes consistency easier to guarantee, as **there is only one place where changes are being made to links**. This means that the card class must call functions in the `account` class to update references, as shown in the implementation of the `changeAccount` operation given below.



```
public class Account
{
   private DebitCard  theCard ;
   public DebitCard getCard() {
      return theCard;
   }
   public void addCard(DebitCard  c) {
      theCard = c;
   }
   public void removeCard() {
     theCard = null;
   }
}
```

```
public class DebitCard
{
   private Account theAccount ;
   public DebitCard(Account a) {
        theAccount = a;
   }
   public Account getAccount() {
     return theAccount;
   }
   public void changeAccount(Account newacc)
   {
        if (newacc.getCard() != null) {
        // throw AccountAlreadyHasACard
        }
        theAccount.removeCard() ;
        newacc.addCard(this) ;
   }
}
```

The Account Class

```
/**
 * The Account class holds a reference variable (at most one) the DebitCard object .
 *  This association is  mutable in both directions. Both Account and DebitCard can change.
 */

   public class Account{

      private String   ownerName;
      private String   accNo;
      private  float   balAmt;

      private    DebitCard    theCard ;
      Account(String name, String accno, float bal){
```

```
        this.ownerName = name;
        this.accNo = accno;
        this.balAmt = bal;
        this.theCard = null;
    }
    public String getOwnerName(){
      return ownerName;
    }
    public String getAccNo(){
      return accNo;
    }
    public float getBalAmt(){
      return balAmt;
    }
    public DebitCard getCard() {
            return theCard;
    }
    public void addCard(DebitCard dcard) {
        // an account object to hold a reference to at most one debit card object
        theCard = dcard;
    }
    public void removeCard() {
    // multiplicity of zero or one variable to hold a null reference.
            theCard = null;
    }
    public String toString(){
        return "Account[ " + accNo + "," + ownerName + "," + balAmt + "]";
    }
}
```

The DebitCard Class

```
/**
 * The DebitCard class holds exactly one Account object reference.
* This association is mutable in both directions. Debit Card can change the Account.
 */
public class DebitCard {

  private String cardno;
  private String issueDate;
  private float amountLimit;
  private Account theAccount;

  public DebitCard( Account acc) {
```

```
        this.theAccount = acc;
    }
    public Account  getAccount() {
      return this. theAccount;
    }
    public  void  changeAccount(Account   newacc) {
        if (newacc.getCard() != null) {
                // throw AccountAlreadyHasACard
        }
        theAccount.removeCard() ;
        newacc.addCard(this) ;
}
    public  void  setCardNo(String  cno){
        this.cardno = cno;
    }
    public  String  getCardNo(){
      return this.cardno;
    }
    public void  setAmtLimit(int  amt){
        this.amountLimit = amt;
    }
    public  float  getAmtLimit() {
      return this.amountLimit;
    }
    public String getIssueDate(){
      return this.issueDate;
    }
    public  void  setIssueDate(String  idate){
      this.issueDate  = idate;
    }
    public  String   toString(){
       return  "DebitCard [ " + cardno + " ," + issueDate +  " , "  + amountLimit + " , "  +
                theAccount.toString()  + " ] " ;
    }
}
```

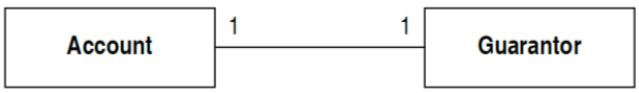
The Test Deriver Class

```
public class TestDebitCard {
  //testing Account and DebitCard association with mutable in both directions.
  public static void main(String[] args) {
        //Create account objects
        Account acc1 = new Account("John Smith", "Acc001", 10000);
        Account acc2 = new Account("Henary Joe", "Acc002", 30000);
```

```
        Account acc3 = new Account("Mary Brown", "Acc003", 50000);

        //Create DebitCard object with Account
        DebitCard dCard1 = new DebitCard(acc1);
        dCard1.setCardNo("DbCard01");   dCard1.setIssueDate( "#12/12/2001#");
        DebitCard dCard2 = new DebitCard(acc2);
        dCard2.setCardNo("DbCard02");  dCard2.setIssueDate("#1/1/2020#");
        DebitCard dCard3 = new DebitCard( acc3);
        dCard3.setCardNo("DbCard03");  dCard3.setIssueDate( "#10/5/2022#");

        //Assign DebitCard  card to Account
        acc1.addCard(dCard1);
        acc2.addCard(dCard2);

        //get the debit card that account is responsible for
        System.out.println(dCard1.toString());
        System.out.println(dCard2.toString());
        System.out.println(dCard3.toString());

// DebitCard[ DbCard01,  #12/12/2001#,  0.00, Account {Acc001, JohnSmith, 10000.00} ]
// DebitCard[ DbCard02,  #1/1/2020#,  0.00, Account {Acc002, Henary Joe, 30000.00} ]
// DebitCard[ DbCard03,  #10/5/2022#,  0.00, Account {Acc003, Mary Brown, 50000.00} ]

        //Change debit card – account
        dCard3.changeAccount(acc3);
        System.out.println(dCard3.toString());
// DebitCard[ DbCard03,  #10/5/2022#,  0.00, Account {Acc003, Mary Brown, 50000.00} ]
   }
}
```

Example 3: Immutable Association in Both Directions

Suppose that the association between **accounts and guarantors is intended to be immutable and needs to be traversed in both directions**. Figure 1.26 shows the relevant class diagram, which preserves the restriction that **each guarantor can only guarantee one account.** Each class can define a data member holding a reference to an object of the other class. These declarations seem to introduce **certain circularity**.



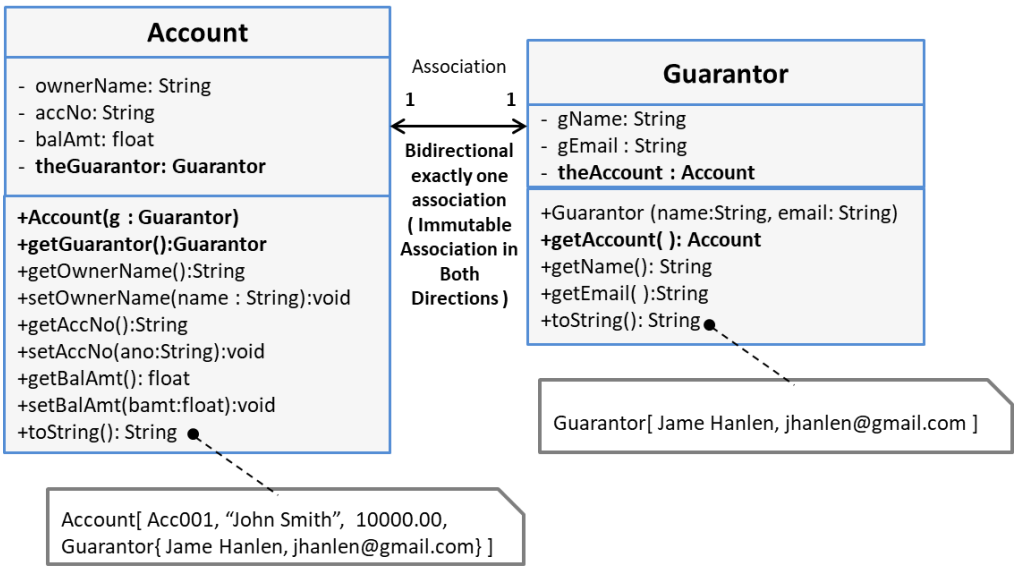


Figure 1.26: An immutable one-to-one association

A better solution is explicitly to **give the responsibility for maintaining the association to one of the classes involved**. In the current case, it is likely that there would be an operation on the guarantor class to create a new account that it grantee for and this would provide a strong argument for making the guarantor class responsible for maintaining the association. **Without the guarantor, no account can be created**. If this was the case, the class declarations may be as follows. For the immutable association in both directions of Account and Guarantor class can be implemented as follows:

```
public class Guarantor {
    ...
 private Account theAccount;
 /* account is initialized in the constructor
 and never modified */

 public Guarantor(String name, String email)
 {
   theAccount = new Account(this);
   gname = name;
   gemail  = email;
 }

 public Account getAccount() {
       return     theAccount ;
 }
 ....
}
```

```
public class Account {
   ...
 private Guarantor theGuarantor;
   /* guarantor is initialized in the
   constructor and never modified */

   public  Account(Guarantor g) {
      if ( g == null ) {
      // Immutable associations
      // throw NullLinkError
      }
      theGuarantor = g ;
   }
   public Guarantor getGuarantor() {
      return theGuarantor ;
   }
   ....
}
```


The declarations in bidirectional implementation introduce a certain condition. **When an account is needed to create, a guarantor must create first**. The guarantor will create an account that guarantee to that account. This could be achieved as shown in the following code.

```
Guarantor g = new Guarantor();
Account a = g.getAccount();
```

In order to create the required link, one of the objects must be created using a default constructor and the link subsequently established with a suitable 'set' operation. It will then be **necessary to check explicitly that the constraints on the association are maintained at all times**.

The Account Class

```
/**
 * The Account class holds a reference variable (exactly one) the Guarantor object.
 * Without the guarantor, no account can be created. Immutable Association in Both Directions.
 */
  public class Account{

     private String   ownerName;
     private String   accNo;
     private  float   balAmt;
     private  Guarantor   theGuarantor;
     public Account(Guarantor   g){
          if ( g == null ) {
            //Immutable associations:  multiplicity of exactly one
                    // throw NullLinkError
          }
          theGuarantor = g ;

     }
     public Guarantor getGuarantor() {
          return  theGuarantor ;
     }
     public  String  getOwnerName(){
        return  ownerName;
     }
     public  void  setOwnerName(String  name){
        this.ownerName = name;
```

```
    }
    public String getAccNo(){
       return accNo;
    }
    public void setAccNo(String acc){
       this. accNo = acc;
    }
    public float getBalAmt(){
       return balAmt;
    }
    public void setBalAmt(float bal){
       this.balAmt = bal;
    }
    public String toString(){
        String str = "Account[ accNo + " , " + ownerName + " , " + balAmt + " , " +
                     theGuarantor.toString() + " ] " ;
        return str;
    }
}
```

The Guarantor Class

```
/**
 * The Guarantor class holds an Account reference.
 * Each guarantor can only guarantee one account. Immutable Association in Both Directions.
 * The responsibility for maintaining the association to the guarantor class to create a new
 * account that it grantee for. Without the guarantor, no account can be created.
 */
public class Guarantor {

   private String name;
   private String email;
   private Account theAccount;

   public Guarantor(String name, String email ) {
        this.name = name;
        this.address = email;
        theAccount = new Account(this);
   }
   public Account getAccount() {
        return this.theAccount ;
   }
   public String getName() {
        return this.name;
```

```
   }
   public String getEmail(){
      return  this.email;
   }
   public String  toString(){
      return  " Guarantor [ " +  name  +  " , "  +  email  +   " ] " ;
   }
}
```

The Test Deriver Class

```
 public class TestGuarantor {
   //testing association between accounts and guarantors is intended to be immutable
   // in both directions.

   public static void main(String[] args) {

         //First create guarantor objects that will automically create each account object
         Guarantor   g1  =  new Guarantor("Jame Hanlen", "jhanlen@gmail.com");
         Guarantor   g2  =  new Guarantor("Horgan Will", "mwill@gmail.com");

         //Get account objects from Guarantor objects and set values
         Account acc1 = g1.getAccount();
         acc1.setOwnerName ("John Smith"); acc1.setAccNo("Acc001");
         acc1.setBalAmt(10000.00);
         Account acc2 = g1.getAccount();
         acc2.setOwnerName("Henary Joe");   acc2.setAccNo("Acc002");
         acc2.setBalAmt( 30000.00);

         Account acc3 = g1.getAccount();
         acc3.setOwnerName("Mary Brown");   acc3.setAccNo("Acc003");
         acc3.setBalAmt(50000.00);

         //get the account that Guarantor is responsible for
         System.out.println(acc1.toString());
         System.out.println(g1.toString());
   //Account[ Acc001, “John Smith”,  10000.00, Guarantor{ Jame Hanlen, jhanlen@gmail.com} ]
   //Guarantor[ Jame Hanlen, jhanlen@gmail.com ]
         System.out.println(acc2.toString());
         System.out.println(g2.toString());
      //Account[ Acc002, “Henary Joe”,  30000.00, Guarantor{ Horgan Will, mwill@gmail.com } ]
      //Guarantor[Horgan Will, mwill@gmail.com ]
   }
}
```

(2) Bidirectional, One-to-Many Association

The bidirectional implementation of one-to-many associations raises no significantly different problems from those discussed above. For example, Figure 1.27 shows an association specifying that **customers can hold many accounts, each of which is held by a single customer**.

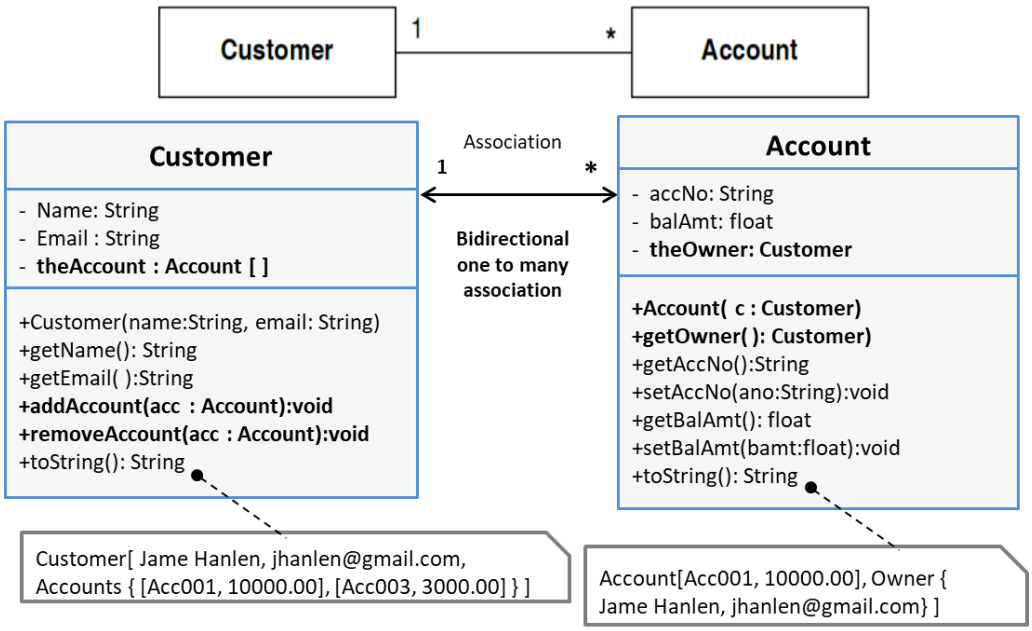


Figure 1.27: Customers holding accounts

As before, the **customer class** could **contain a data member** to **store a collection of pointers to accounts**, and **additionally each account should store a single pointer to a customer**. It would seem most sensible to **give the customer class the responsibility of maintaining the links of this association**, though in practice this decision would only be made in the light of the total processing requirements of the system.

```
public class Customer {
 private List<Account>  theAccounts ;
 public Customer(){
 theAccounts = new ArrayList<Account>();
 }
 public void addAccount(Account  acc) {
   if (!theAccounts.contains(acc)) {
        theAccounts.addElement(acc) ;
    }
 }
```

```
public class Account {
 private  Customer  theOwner;

 public  Account(Customer c) {
    if (c == null) {
      // throw NullCustomerError
    }
    theOwner = c ;
  }

 public Customer getOwner() {
```

```
  public void removeAccount(Account acc) {
    theAccounts.removeElement(acc);
  }
 }
```

```
    return   this.theOwner ;
  }
 }
```

The implementation sketched below follows the strategy allocating the responsibility of manipulating references exclusively to the customer class.

The Account Class

```
/**
 * The Account class holds a reference variable (exactly one) the Customer object.
 * An immutable association with multiplicity 'one' Customer object and a mutable association
 * with multiplicity 'many'  Account objects.
 */
  public class Account{

     private  String   accNo;
     private  float   balAmt;
     private  Customer  theOwner;

     public  Account(Customer c) {
       if (c == null) {
         // throw NullCustomerError
       }
       theOwner = c ;
      }
     public Customer getOwner() {
       return   this.theOwner ;
     }
     public  String  getAccNo(){
        return  this.accNo;
     }
     public  void  setAccNo(String   acc){
        this.accNo =  acc;
     }
     public  float  getBalAmt(){
        return  balAmt;
     }
     public  void  setBalAmt(float   bal){
        this.balAmt =  bal;
     }
     public String  toString(){
          String str  =  "Account[ accNo  +  " , "  +  balAmt  +  " ,  Owner { "  +
```

```
                    theOwner.getName() + "," + theOwner.getEmail() + "}]" ;
        return str;
    }
}
```

The Customer Class

```
/**
 * The Customer class holds multiple Account objects of its own.
 * The customer class takes the responsibility of maintaining the links of this association,
 */
public class Customer {

   private String name;
   private String email;
   private List<Account> theAccounts ;

   public Customer(String name, String email ) {
        this.name = name;
        this.email = email;
        theAccount = new ArrayList<Account>();
   }

   public void addAccount(Account acc) {
     if (!theAccounts.contains(acc)) {
         theAccounts.addElement(acc) ;
     }
   }
    public void removeAccount(Account acc) {
          theAccounts.removeElement(acc) ;
    }
   public String getName() {
        return this.name;
   }
   public String getEmail(){
     return this.email;
   }
   public String toString(){
      String str = "Customer [" + name + "," + email + ", Accounts {" ;
       for(Account acc : theAccount){
            str = str + "[" + acc.getAccNo() + "," + acc.getBalAmt() + "],";
       }
       str = str + "}]" ;
       return str;
   }
```

```
}
```

The Test Deriver Class

```
public class TestCustomer {
  //testing association between account  and customer

  public static void main(String[] args) {

        //First create customer objects
        Customer   c1  =  new Customer("Jame Hanlen", "jhanlen@gmail.com");
        Customer   c2  =  new Customer("Horgan Will", "mwill@gmail.com");

        //Create account object  with customer object and set values
        Account acc1 = new Account (c1);
        acc1.setAccNo("Acc001");  acc1.setBalAmt(10000.00);
        Account acc2 = new Account(c1);
        acc2.setAccNo("Acc002");  acc2.setBalAmt( 30000.00);
        Account acc3 = new Account(c2);
        acc3.setAccNo("Acc003"); acc3.setBalAmt( 50000.00);

        //assign Customer and Account that account is responsible for
      // the customer class  takes the responsibility of maintaining the links of this association
        c1.addAccount(acc1);
        c1.addAccount(acc2);
        c2.addAccount(acc3);

        System.out.println(c1.toString());
        System.out.println(c2.toString());
  //Customer[ Jame Hanlen, jhanlen@gmail.com, Accounts{ [Acc001, 10000.00],
  // [Acc002, 30000.00]} ]
  //Customer[Horgan Will, mwill@gmail.com, Accounts{ [ Acc003, 50000.00] } ]

        System.out.println(acc1.toString());
        System.out.println(acc2.toString());
        System.out.println(acc3.toString());

     //Account[ Acc001, 10000.00, Owner{ Jame Hanlen, jhanlen@gmail.com } ]
     //Account [Acc002, 30000.00, Owner{ Jame Hanlen, jhanlen@gmail.com }]
     //Account [Acc003, 50000.00, Owner{ Horgan Will, mwill@gmail.com }]

  }
}
```

(3) Bidirectional, Many-to-Many Association

The most general case of many-to-many associations does not raise any new issues of principle. For example, consider the relationship in Figure 1.28, which allows a number of signatories (participants), **people who are authorized to sign cheques**, to be defined **for each account.** At the same time, **a person can be a signatory for any number of accounts. This association may need to be traversed in both directions, and is mutable at both ends**.

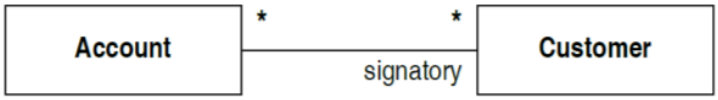


Figure 1.28: A general many-to-many association

There is no reason in principle why **this association should not be implemented using the techniques discussed above**. **Because of the symmetry of many-to-many associations**, however, it is often **difficult sensibly to assign the responsibility of maintaining the links to one of the two classes involved and it can be quite complex to ensure that the operations in each class correctly maintain the referential integrity of the links.**

The following code provides an outline **implementation of the association** in Figure 1.28 where **responsibility for maintaining the links between objects is given to the `account` class.**

```
public class Customer {
  private Vector theSignatoryAccounts ;
  public Customer() {
    theSignatoryAccounts = new Vector() ;
  }
  public void addSignatoryAccount(Account a)
  {
    theSignatoryAccounts.addElement(a) ;
  }
}
```

```
public class Account {
  private Vector theSignatories ;
  public Account() {
    theSignatories = new Vector() ;
  }
  public void addSignatory(Customer c) {
    if (!theSignatories.contains(c)) {
        theSignatories.addElement(c) ;
        c.addSignatoryAccount(this) ;
    }
  }
}
```

An alternative approach that can sometimes be useful is to apply the **technique of reifying the association**. In the present case, this would involve **replacing the many-to-many association by a new class and a pair of one-to-many associations**, as shown in Figure 1.29.

Figure 1.29: Reifying a many-to-many association

Now the **responsibility of maintaining the signatory relationship can be given to the new signatory class**. Each instance of this class maintains references to the linked account and customer objects and should be made responsible for ensuring that the inverse references are maintained consistently.

The following code adds a signatory class, as shown in Figure 1.29, but keeps the original interface whereby **the account class is responsible for maintaining links between customers and accounts.** In this case, however, it does it indirectly by creating a new signatory object. In this case, where a signatory class is introduced, the changes required are more complex. The `addSignatory' method must iterate through the link signatories, checking the customer of each, and only proceed if the customer being added does not appear in any of the signatory instances:

```
public class Customer {

  private Vector theSignatories ;

  public Customer() {
    theSignatories = new Vector() ;
  }

  public void addSignatory(Signatory s) {
    theSignatories.addElement(s) ;
  }
}
```

```
public class Account {

  private Vector theSignatories ;

  public Account() {
    theSignatories = new Vector() ;
  }

public void addSignatory(Customer   c) {
    boolean  newCustomer = true ;
    Enumeration enum =
theSignatories.elements();
    while (newCustomer &&
                  enum.hasMoreElements()) {
      Signatory sig = (Signatory)
enum.nextElement();
      if (sig.getCustomer() == c) {
                newCustomer = false ;
      }
    }  //while end
    if (newCustomer) {
      Signatory s = new Signatory(c, this) ;
      theSignatories.addElement(s) ;
      c.addSignatory(s) ;
    }
  }
}
```

```
public class Signatory  {
  private   Customer   theCustomer ;
```

```
  private  Account  theAccount ;

  public  Signatory(Customer  c, Account  a) {
   if (c == null || a == null) {
      // throw NullLinkError
   }
   theCustomer = c ;
   theAccount = a ;
  }

  public Customer  getCustomer() {
   return  theCustomer ;
  }
}
```

The implementation code shows how signatory class replace many-to-many association relationship between Account class and Customer class for a number of signatories (participants), **people who are authorized to sign cheques**, to be defined **for each account.** At the same time, **a person can be a signatory for any number of accounts. This association may need to be traversed in both directions, and is mutable at both ends**.

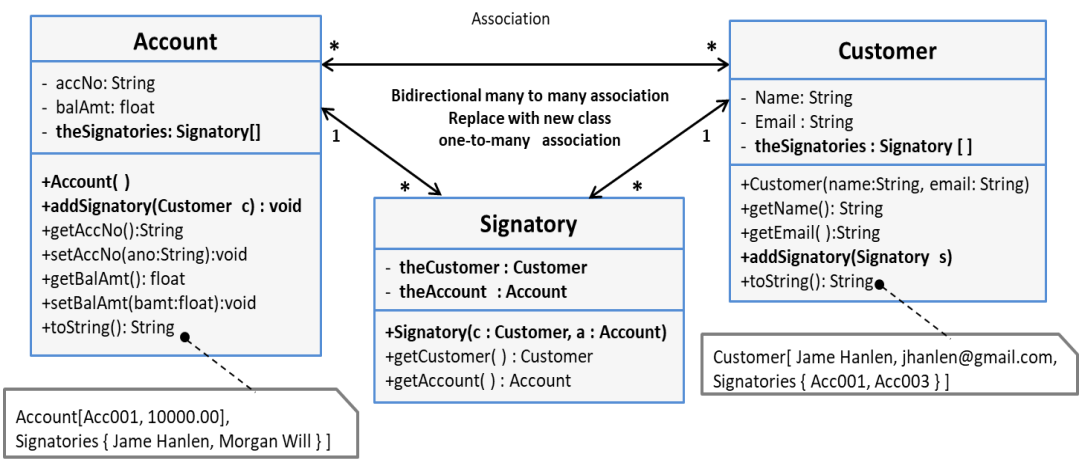


Figure 1.30: Reifying a many-to-many association to one-to-many association with a new class

The Account Class

```
/**
 * The Account class holds a number of signatories (participants) objects.
 * A mutable association at both ends and traverse in both directions
 * The association with multiplicity 'one-to-many' Account object and Customer objects.
 * The account class is responsible for maintaining links between customers and accounts.
```

```
 */
public class Account {

   private  String   accNo;
   private  float    balAmt;
   private List<Signatory>  theSignatories ;

    public Account(String  ano, float  amt ) {
        this.accNo  =  ano;
        this.balAmt  =  amt;
        this.theSignatories = new  ArrayList<Signatory>( ) ;
    }
   //   The `addSignatory' method must iterate through the link signatories,
   // checking the customer of each, and only proceed if the customer being added
   // does not appear in any of the signatory instances:

     public  void  addSignatory(Customer   c) {
           boolean  newCustomer = true ;
           Enumeration enum = theSignatories.elements();
           while (newCustomer &&  enum.hasMoreElements()) {
              Signatory   sig  =  (Signatory) enum.nextElement();
                 if (sig.getCustomer() == c) {
                    newCustomer = false ;
                 }
            }  //while end

           if (newCustomer) {
                Signatory  s  = new Signatory( c,  this ) ;
                theSignatories.addElement(s) ;
                c.addSignatory(s) ;
              }
     }
     public  String  getAccNo(){
        return  this.accNo;
     }
     public  void  setAccNo(String   acc){
        this.accNo =  acc;
     }
     public  float  getBalAmt(){
        return  balAmt;
     }
     public  void  setBalAmt(float    bal){
        this.balAmt =  bal;
     }
```

```
    public String toString(){
        String str = "Account[ accNo + " , " + balAmt + " , Signatories { " ;
        for(Signatory s : theSignatories){
            str = str + s.getCustomer().getName() + " , " ;
        }
        str = str + " }] " ;
        return str;
    }
}
```

The Customer Class

```
/**
 * The Customer class holds multiple Signatories objects
 * who are authorized to sign cheques defined for each account
 */

public class Customer {

   private String name;
   private String email;
   private List<Signatory> theSignatories ;

   public Customer(String name, String email) {
        this.name = name;
        this.email = email;
        theSignatories = new ArrayList<Signatory>( ) ;
   }
   public void addSignatory(Signatory s) {
       theSignatories.addElement(s) ;
   }
   public String getName() {
        return this.name;
   }
   public String getEmail(){
     return this.email;
   }
   public String toString(){
      String str = " Customer [ " + name + " , " + email + " , Accounts { " ;
       for(Signatory s : theSignatories){
            str = str + s.getAccount().getAccNo() + " , " ;
       }
       str = str + " } ] " ;
       return str;
   }
```

```
}
```

The Signatory Class

```
/**
 * The Signatory class holds a Customer object and an Account object.
 * The Customer who is authorized to sign cheques defined for each account
 */

public class Signatory  {
 private   Customer   theCustomer ;
 private   Account   theAccount ;

 public Signatory(Customer  c, Account  a) {
     if (c == null || a == null) {
          // throw NullLinkError
      }
      theCustomer = c ;
      theAccount = a ;
 }

 public Customer  getCustomer() {
  return  theCustomer ;
 }
 public  Account  getAccount() {
  return  theAccount ;
 }
}
```

The Test Deriver Class

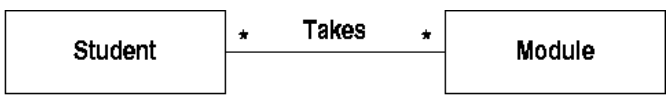
```
 public class TestCustomer {
  //testing many-to-many association between account  and customer

  public static void main(String[] args) {

        //First create customer objects
        Customer   c1  =  new Customer("Jame Hanlen", "jhanlen@gmail.com");
        Customer   c2  =  new Customer("Morgan Will", "mwill@gmail.com");

        //Create account object  with customer object and set values
        Account acc1 = new Account ("Acc001",  10000.00);
        Account acc2 = new Account( "Acc002", 30000.00);
        Account acc3 = new Account( "Acc003", 50000.00);

        //assign the Customer who is authorized to sign cheques defined for each account
```

```
        // the Account class  takes the responsibility of maintaining the links of this association
          acc1.addSignatory(c1);
          acc1.addSignatory(c2);
          acc2.addSignatory(c1);
          acc3.addSignatory(c1);

          System.out.println(c1.toString());
          System.out.println(c2.toString());
    //Customer[ Jame Hanlen, jhanlen@gmail.com, Signatories{ Acc001, Acc002,Acc003}]
    //Customer[Morgan Will, mwill@gmail.com, Signatories{ Acc001 } ]

          System.out.println(acc1.toString());
          System.out.println(acc2.toString());
          System.out.println(acc3.toString());

       //Account[ Acc001, 10000.00, Signatories{ Jame Hanlen, Morgan Will } ]
       //Account [Acc002, 30000.00, Owner{ Jame Hanlen }]
       //Account [Acc003, 50000.00, Owner{ Jame Hanlen }]
    }
}
```

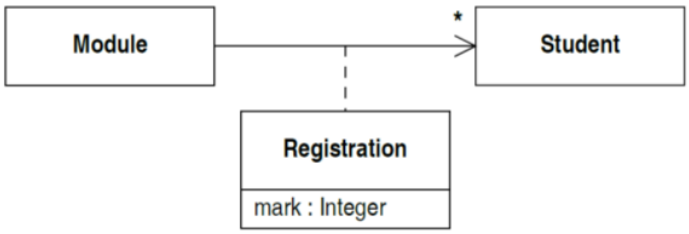
3.5.8 Implementing Association Classes

Some **association links** can be **described with the attributes**. Consider, for example, the **association** between **students** and the **modules**, the class diagram shown in figure 1.31(a) models that a student takes a module and system needs to record all the marks for modules that students took. Here the 'mark' attribute only makes sense if student takes the module and student may take many modules and it is necessary to record more than one mark for each student. So it's not simply an attribute of either class of Student and Module. Therefore **'mark' attribute is associated with the link between two objects** rather than with either of the individual objects.

Association class is shown as an association and a class icon, linked by a dashed line. Figure 1.31(b) shows the association class enabling a single mark to be recorded every time a student takes a module. This replaces the association "Takes" defined in figure 1.31(a).



(a) A simple model for student takes modules.



(b) An association class to store 'marks' attributes for both class association
Figure 1.31: Association class example

Association classes provide a means of **associating data values with links**. An association class is a single model element in UML, one which **has all the properties of both an association and a class**. It **connects two classes having attributes** and **store data belonging** specifically to the link. **Association classes share the properties of associations** and classes they can define links between objects and they **allow attribute values to be stored**.

Figure 1.32: Transforming the association into a class

A further consideration in the implementation of this association is that the two class diagrams in Figure 1.31 and Figure 1.32 in fact have slightly different meanings. Figure 1.31 states that a student can only take a module once, as only one link is permitted between any given pair of module and student objects. With Figure 1.32 on the other hand, there is nothing to prevent a student being linked to the same module many times, through the mediation of different registration instances. An implementation of Figure 1.32 should bear this in mind and check that the appropriate constraints are satisfied.

A common strategy in this case is to transform the association class into **a simple class linked to the two original classes with two new associations**, as shown in Figure 1.32. In this diagram, the fact that **many students can take a module** is modeled by stating that the module can be linked to many objects of the registration class, each of which is further linked to a unique student, namely the student for whom the registration applies.

A possible outline implementation for this operation is shown below. The implementation of the registration class is very simple. It must **store a reference to the linked student object and the mark** gained by that student. As this class is manipulated exclusively by the module class, we do not bother to provide an operational interface for

it. The relevant parts of the definition of the **module class** are sketched out below. The very simple implementation given in this example is given below.

```
class Registration  {

   private  Student   student;
   private int  mark ;

   Registration(Student st) {
         student = st;
         mark = 0;
   }
   ……
}
```

```
public class Module {

   private  Vector  registrations ;

   public void enroll (Student  st)
   {
      registrations.addElement (
               new Registration(st));
   }
   ….
}
```

For the case of the association class in Figure 1.31, the natural approach seems to be to allow the **registration class to maintain the links between registrations, students and modules.** For example, these links will be created when a student registers for a module, and so will be created when a registration object is created. The following code defines the classes with methods permitting a class list to be printed off for a module, and a list of modules taken for a student, thus demonstrating that the links can be traversed in both directions.

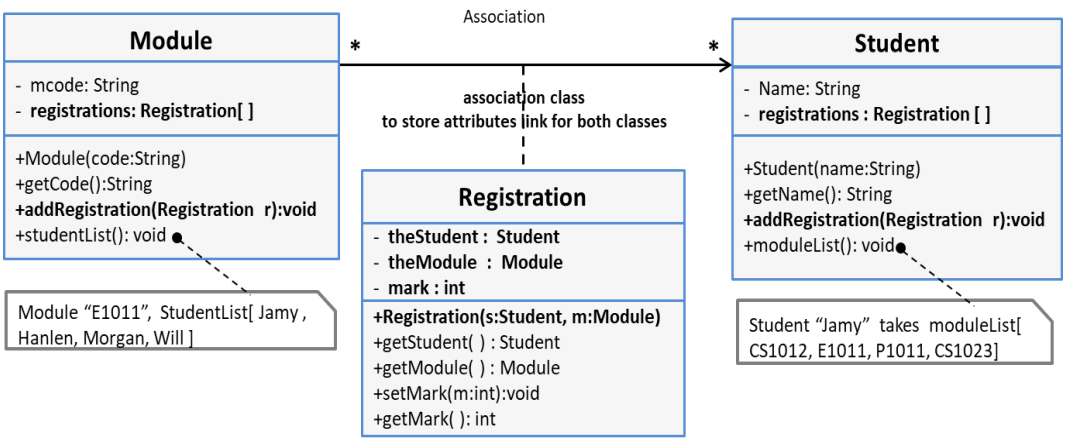


Figure 1.33: The association class – Registartion maintain the links between student and module class

The Student Class

```
/*  The student class models for a student takes a module and
  * to record all the marks for modules that students took . */
```

```
public class Student  {

 private  String name ;
 private  List<Registration>  registrations = new  ArrayList<Registration>( ) ;

 public Student(String n) {
    name = n ;
 }
 public String getName() {
    return name ;
 }
 public void addRegistration(Registration  r) {
   registrations.add(r) ;
 }
 public void  ModuleList() {
    Enumeration enum = registrations.elements() ;
  System.out.println("Student  "  +  name  +  " takes modulelist [ ") ;
  while (enum.hasMoreElements()) {
      Registration r = (Registration) enum.nextElement() ;
      System.out.print(r.getModule().getCode()  +  " ,  ") ;
  }
  System.out.println("  ] ");
 }
 }
```

The Module Class

```
/* The module class  */
public class Module {
   private  String  mcode ;
   private  List<Registration>  registrations = new ArrayList<Registration>( ) ;

 public Module(String c) {
    mcode = c ;
 }
 public String getCode() {
    return  mcode ;
 }
 public void addRegistration(Registration r) {
    registrations.add(r) ;
 }
 public void classList() {
    Enumeration enum = registrations.elements() ;
    System.out.print("Module – "   +  mcode +  " ,  Student List [" );
    while (enum.hasMoreElements()) {
```

```
        Registration r = (Registration) enum.nextElement() ;
        System.out.print(r.getStudent().getName() +  " , " ) ;
    }
    System.out.println("  ]  ") ;
 }
 }
```

The Registration Class

```
/* The registration class maintain the links between registrations, students and modules. */
public class Registration {

   private  Student  theStudent ;
   private  Module  theModule ;
   private   int    mark ;

  public Registration(Student s, Module m) {
     theStudent = s ;
     s.addRegistration(this) ;
     theModule = m ;
     m.addRegistration(this) ;
  }
  public Student getStudent() {
      return theStudent ;
  }
  public Module getModule() {
     return theModule ;
  }
  public void setMark(int m){
    mark = m;
  }
 public int  getMark(){
    return  mark;
 }
}
```

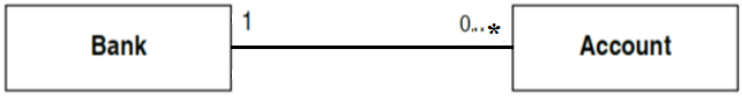
The Test Deriver Class

```
 public class TestRegistration {
   //testing association class registration that maintain links between student  and module

   public static void main(String[] args) {

         //create student objects
```

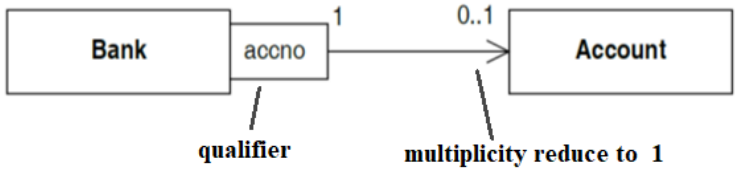
```
        Student   s1 = new  Student("Jamy");
        Student   s2 = new  Student("Halen");
        Student   s3 = new  Student("Morgan");
        Student   s4 = new  Student("Will");

        //create module objects
        Module  m1 = new Module("E1011");
        Module  m2 = new Module("M1011");
        Module  m3 = new Module("P1011");
        Module  m4 = new Module("CS1012");
        Module  m5 = new Module("CS1023");

        //make registration with student and module
        Registration   r1 = new  Registration(s1,m1);
        Registration   r2 = new  Registration(s1,m2);
        Registration   r3 = new  Registration(s2,m1);
        Registration   r4 = new  Registration(s2,m2);
        Registration   r5 = new  Registration(s2,m3);
        Registration   r6 = new  Registration(s3,m1);
        Registration   r7 = new  Registration(s3,m3);
        Registration   r8 = new  Registration(s4,m1);
        Registration   r9 = new  Registration(s4,m3);
        Registration   r10 = new  Registration(s4,m5);

        //display each module registred student list
        m1.studentList(); // Jamy, Halen, Morgan, Will
        m2.studentList(); // Jamy, Halen
        m3.studentList(); // Halen, Morgan, Will
        m4.studentList(); // ..
        m5.studentList(); // Will

        //display each student registred module list
        s1.moduleList(); // E1011, M1011
        s2.moduleList(); // E1011, M1011, P1011
        s3.moduleList(); // E1011, P1011
        s4.moduleList(); // E1011, P1011, CS1023
    }
    }
```

3.5.9 Qualified Association

The previous sections have discussed about the implementation of simple associations. The **qualified association** is **a specialized form of associations** where a **qualifier** is key data that is used to select an object (or objects) from a larger set of related objects, based upon the qualifier key. The most common use of such an association is to provide **efficient access to objects based on the values of the qualifying attributes**. For example, Figure 1.34(a) shows an association modeling the fact that **a bank can maintain many different accounts**. In Figure 1.34(b), e**ach account** is identifiable by a single piece of data, namely the **account number (accno)**, and this attribute is shown as **a qualifier attached to the bank class**.



(a) A bank can maintain many different accounts



(b) Each account can be selected by the qualifier key - accno

Figure 1.34: A qualified association for a bank class to identify an account

There is one subtle point about qualified associations: the change in multiplicity. For example, as shown in Figure 1.34(a) vs. (b), **qualification reduces the multiplicity** at the target end of the association, usually **down from many to one**, because it implies the selection of usually **one instance from a larger set**.

For simplicity, we assume that the association is to be given a unidirectional implementation. For example, to **retrieve information about accounts given only an account number**, if the bank simply held a pointer to each of its accounts, this operation could be implemented by searching through all the pointers until the matching account was found. This could be very slow, however, and **a better approach** might be to maintain **a lookup table mapping account numbers to accounts**, as illustrated in Figure 1.35. As account numbers can be ordered, for example, this gives the possibility of using much more efficient search techniques. This kind of structure is relatively easy to implement, the major issue being to decide how to implement the lookup table. **In Java**, an obvious and straightforward choice is to use **a utility class** such as **`java.util.HashTable`**.

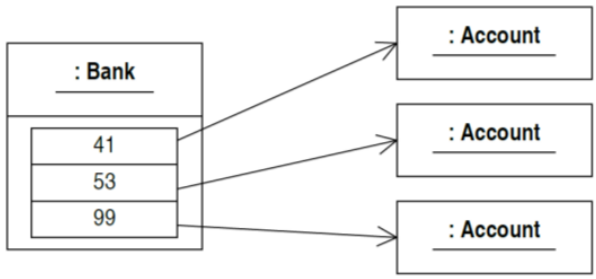


Figure 1.35: How qualifiers can be implemented using a lookup table mapping account numbers to accounts

As **a unidirectional implementation** is being considered, shown in Figure 1.34, the **bank object must have the responsibility of maintaining the association**. Operations to add and remove accounts, and to look up an account given its number, could be declared as follows.

```
public class Bank
{
    private Hashtable theAccounts ;
    public void addAccount(Account a) {
        theAccounts.put(new Integer(a.getNumber()), a);
    }
    public void removeAccount(int accno) {
        theAccounts.remove(new Integer(accno));
    }
    public  Account  lookupAccount(int accno) {
        return (Account) theAccounts.get(new Integer(accno));
    }
}
```

Qualifiers, then, can still be handled using the simple model that implements **individual links by references**. The implementation given above treats a qualified association in much the same way as **an association with multiplicity 'many'**. The most **significant difference** is in the **data structure used to hold the multiple pointers at one end of the association.**

In the case of the qualified association between bank and accounts, shown in Figure 1.34(b), it is likely that the Bank class would have the responsibility for maintaining the links between objects. The code below contains a method in the Bank class to create an account; the bank field in the new account is thereby filled by the bank creating it.

The Account Class

```
/* The Account class with qualifier account number (accno)
 *  The qualified association with the Bank class
 */
   public class Account
   {
    private  int  accno ;
    private  Bank  theBank ;
    private  float   balance;

    public Account(int n, Bank b) {
          accno = n ;
          theBank = b ;
          balance = 10000.00;
    }
    public int getNumber() {
          return accno ;
    }
    public  float  getBalance()[
         return   balance;
    }
    public Bank getBank() {
          return theBank ;
    }
   }
```

The Bank Class

```
/* The Bank class qualified associations with the Account class in multiplicity 'many'.
 * The Bank class maintains a lookup table mapping account numbers to accounts.
 */
   public class Bank
   {
    private  String  name;
    private Hashtable theAccounts ;

    public Bank (String   n){
       name  =  n;
       theAccounts = new Hashtable() ;
    }
    public  String  getName(){
         Return name;
    }
```

```
    public  void  createAccount(int   n) {
        addAccount(new Account(n, this)) ;
    }
    public void addAccount(Account    a) {
        theAccounts.put(new Integer(a.getNumber()), a);
    }
    public void removeAccount(int accno) {
        theAccounts.remove(new Integer(accno));
    }
    public   Account    lookupAccount(int accno) {
         return  (Account) theAccounts.get(new Integer(accno));
    }
  }
```

The Test Deriver Class

```
public class TestQualified {
  //testing qualified association links between Account  and  Bank.

  public static void main(String[] args) {
        //Create Bank object
        Bank  aBank = new Bank("A Bank" );

        //Create Account objects
        aBank.createAccount(001);
        aBank.createAccount(002);
        aBank.createAccount(003);

        //Display  list of Accounts in a Bank
        System.out.println("List of Accounts at:"  +  aBank.getName() );
        Account   acc = aBank.lookupAccount(003);
        System.out.println("Account Number: "  +  acc.getNumber()  +  " , "  +  acc.getBalance() );

        // List of Accounts at: A  Bank
        // Account Number:   003 , 10000.00
  }
```

A bidirectional implementation of a qualified association, therefore, does not raise any significantly new problems and implementation of the association in Figure 1.19 as **a bidirectional association is left as an exercise.**

3.5.10 Difference between Association, Aggregation, Composition in Java

Association is a relation between two separate classes which is established through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many. We have learned association implementation in above section. In Object-Oriented programming, an Object communicates to another object to use functionality and services provided by that object. Aggregation(HAS-A) and Composition(Belongs-to) are the two forms of association.

Association in java is one of the types of relations between classes. Aggregation is a relatively weak association, whereas Composition is a strong association. Composition can be called a more restricted form of Aggregation. Aggregation can be called the superset of Composition, since all Compositions can are Aggregations but, not all Aggregations can be called Composition.

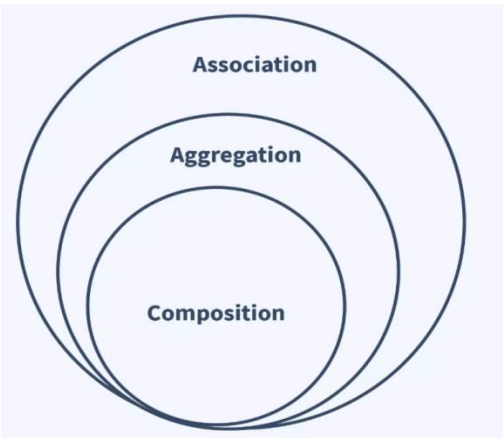


Figure 1.xx: Association, Aggregation and Composition relation

Association, Aggregation and Composition are three kinds of relationships between classes. They are related with design principles. There are a million ways to use OOP to devise solutions that best suit your needs. These solutions may not work in every case. That's precisely you must have a whole arsenal of OOP related concepts so that, when devising a solution, you can use that arsenal and devise a perfect solution that best suits your need. These three concepts are explained with the example as follows:

Association - I have a relationship with an object. *Foo* uses *Bar*

```
public class Foo {
   private Bar bar;
};
```

Composition - I own an object and I am responsible for its lifetime. When *Foo* dies, so does *Bar*.

```
public class Foo {
   private Bar bar = new Bar();
};
```

Aggregation - I have an object which I've borrowed from someone else. When Foo dies, Bar may live on.

```
public class Foo {
   private Bar bar;
   Foo(Bar bar) {
        this.bar = bar;
   }
}
```

By carefully understanding the diagram above and the examples provided above we can conclude that

- **association** is the **weakest relationship** between classes,
- **aggregation** is **somewhat stronger relationship** than association and
- **composition** is the **strongest relationship**

We are going to code program that uses these relationships.
Source: https://www.geeksforgeeks.org/association-composition-aggregation-java/

Association examples:

- Two separate classes ***Bank*** and ***Employee*** are associated through their Objects. Bank can have many employees, So, it is a one-to-many relationship.
- Similarly, every ***city*** exists in exactly one ***state***, but a state can have many cities, which is a "many-to-one" relationship.
- Lastly, the association between a ***teacher*** and a ***student***, multiple students can be associated with a single teacher and a single student can also be associated with multiple teachers but both can be created or deleted independently. This is a "many-to-many" relationship.

```
// Java Program to illustrate the Concept of Association

// Importing required classes
import java.io.*;
import java.util.*;
```

```
// Class 1 : Bank class
class Bank {

   // Attributes of bank
   private String bankName;
   private Set<Employee> employees;

   // Constructor of Bank class
   public Bank(String bankName)   {
      this.bankName = bankName;
   }

   // Method of Bank class
   public String getBankName()  {
      return this.bankName;
   }

   public void setEmployees(Set<Employee> employees)  {
      this.employees = employees;
   }

   public Set<Employee> getEmployees()  {
      return this.employees;
   }
}

// Class 2 : Employee class
class Employee {

   // Attributes of employee
   private String name;

   // Constructor of Employee class
   public Employee(String name)   {
      this.name = name;
   }

   // Method of Employee class
   public String getEmployeeName()  {
       return this.name;
   }
}
```

```
// Class 3 : Association between both the  classes in main method
class AssociationExample {

   // Main driver method
   public static void main(String[] args)
   {
      // Creating Employee objects
      Employee emp1 = new Employee("Ridhi");
       Employee emp2 = new Employee("Vijay");

       // adding the employees to a set
      Set<Employee> employees = new HashSet<>();
      employees.add(emp1);
      employees.add(emp2);

       // Creating a Bank object
       Bank bank = new Bank("ICICI");

       // setting the employees for the Bank object
       bank.setEmployees(employees);

       // traversing and displaying the bank employees
       for (Employee emp : bank.getEmployees()) {
          System.out.println(emp.getEmployeeName()
                        + " belongs to bank "
                        + bank.getBankName());
      }
   }
}
```

Aggregation is a unidirectional association i.e. **a one-way relationship**. For example, a ***department*** can have ***students*** but vice versa is not possible and thus uni-directional in nature. However both entries can survive individually which means ending one entity will not affect the other entity. The main reason why you need **Aggregation is to maintain code reusability**.

Example, There is an Institute which has no. of departments like CSE, EE. Every department has no. of students. So, we make an Institute class that has a reference to Object or no. of Objects (i.e. List of Objects) of the Department class. That means Institute class is associated with Department class through its Object(s). And Department

class has also a reference to Object or Objects (i.e. List of Objects) of the Student class means it is associated with the Student class through its Object(s).

```
// Java program to illustrate - Concept of Aggregation

// Importing required classes
import java.io.*;
import java.util.*;

// Class 1 : Student class
class Student {

   // Attributes of Student
   private String studentName;
   private int studentId;

   // Constructor of Student class
   public Student(String studentName, int studentId)    {
      this.studentName = studentName;
      this.studentId = studentId;
   }

   public int getstudentId() {
    return studentId;
   }

   public String getstudentName() {
    return studentName;
   }
}

// Class 2 : Department class
// Department class contains list of Students
class Department {

   // Attributes of Department class
   private String deptName;
   private List<Student> students;

   // Constructor of Department class
   public Department(String deptName, List<Student> students)   {
      this.deptName = deptName;
      this.students = students;
```

```
    }
    public List<Student> getStudents() {
      return students;
    }

    public void addStudent(Student student)  {
        students.add(student);
    }
}

// Class 3 : Institute class
// Institute class contains the list of Departments
class Institute {

    // Attributes of Institute
    private String instituteName;
    private List<Department> departments;

    // Constructor of Institute class
    public Institute(String instituteName, List<Department> departments)    {
        this.instituteName = instituteName;
        this.departments = departments;
    }

    public void addDepartment(Department department)  {
        departments.add(department);
    }

    // Method of Institute class
    // Counting total students in the institute
    public int getTotalStudentsInInstitute()  {
        int noOfStudents = 0;
        List<Student> students = null;

        for (Department dept : departments) {
            students = dept.getStudents();
            for (Student s : students) {
                noOfStudents++;
            }
        }
        return noOfStudents;
    }
}
```

```
// Class 4 : main class
class AggregationExample {
   // main driver method
   public static void main(String[] args)    {
      // Creating independent Student objects
      Student s1 = new Student("Parul", 1);
      Student s2 = new Student("Sachin", 2);
      Student s3 = new Student("Priya", 1);
      Student s4 = new Student("Rahul", 2);

      // Creating an list of CSE Students
      List<Student> cse_students = new ArrayList<Student>();
      cse_students.add(s1);
      cse_students.add(s2);

      // Creating an initial list of EE Students
      List<Student> ee_students = new ArrayList<Student>();
      ee_students.add(s3);
      ee_students.add(s4);

      // Creating Department object with a Students list
      // using Aggregation (Department "has" students)
      Department CSE = new Department("CSE", cse_students);
      Department EE = new Department("EE", ee_students);

      // Creating an initial list of Departments
      List<Department> departments = new ArrayList<Department>();
      departments.add(CSE);
      departments.add(EE);

      // Creating an Institute object with Departments list
      // using Aggregation (Institute "has" Departments)
      Institute institute = new Institute("BITS", departments);

      // Display message for better readability
      System.out.print("Total students in institute: ");

      // Calling method to get total number of students
      // in the institute and printing on console
      System.out.print(
         institute.getTotalStudentsInInstitute());
   }
}
```

Composition is a restricted form of Aggregation in which **two entities are highly dependent on each other.** It represents **part-of** relationship. In composition, both entities are dependent on each other. When there is a composition between two entities, the composed object cannot exist without the other entity.

It simply means that **one of the objects is a logically larger structure**, which contains the other object. It means that **if we destroy the owner object, its members also will be destroyed with it.** For example, if the building is destroyed the room is destroyed. But, note that it doesn't mean, that the containing object can't exist without any of its parts. For example, if we tear down all the rooms inside a building, the building will still exist.

Example, a company can have no. of departments. All the departments are part-of the Company. So, if the Company gets destroyed then all the Departments within that particular Company will be destroyed, i.e. Departments can not exist independently without the Company. That's why it is composition. Department is Part-of Company.

```
// Java program to illustrate Concept of Composition
// Importing required classes
import java.io.*;
import java.util.*;

// Class 1 : Department class
class Department {

   // Attributes of Department
   public String departmentName;
   // Constructor of Department class
   public Department(String departmentName)  {
      this.departmentName = departmentName;
   }
   public String getDepartmentName()  {
      return departmentName;
   }
}

// Class 2 : Company class
class Company {

   // Reference to refer to list of books
   private String companyName;
   private List<Department> departments;
```

```
    // Constructor of Company class
    public Company(String companyName)    {
        this.companyName = companyName;
        this.departments = new ArrayList<Department>();
    }

    // Method to add new Department to the Company
    public void addDepartment(Department department)   {
        departments.add(department);
    }
    public List<Department> getDepartments()   {
        return new ArrayList<>(departments);
    }

    // Method to get total number of Departments in the Company
    public int getTotalDepartments()   {
        return departments.size();
    }
}

// Class 3: Main class
class CompositonExample {

    // Main driver method
    public static void main(String[] args)    {
        // Creating a Company object
        Company techCompany = new Company("Tech Corp");

        techCompany.addDepartment(new Department("Engineering"));
        techCompany.addDepartment(new Department("Operations"));
        techCompany.addDepartment(new Department("Human Resources"));
        techCompany.addDepartment(new Department("Finance"));

        int d = techCompany.getTotalDepartments();
        System.out.println("Total Departments: " + d);

        System.out.println("Department names: ");
        for (Department dept : techCompany.getDepartments()) {
            System.out.println("- " + dept.getDepartmentName());
        }
    }
}
```

Below are the primary differences between the forms of Association, Composition and Aggregation in java:

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

Understanding how the different forms of association work give us the ability to write code that is closely relevant and practical in the real world.

Check Your Understanding on Association Relationship between Classes

Q1. What do you understand by the term 'Association' in UML? Discuss the types of associations. How does Association differ from Link?

Q2. The association: "Each department offers one or more courses" will have the following multiplicity at the course (child) end of the association.

(a) 1..1

(b) 0..*

(c) 0..1

(d) 1..*

Q3. For the association "Each automobile may be owned by at most one driver", the multiplicity at the driver (parent) end of the relationship will be:

(a) 0..1

(b) 1..1

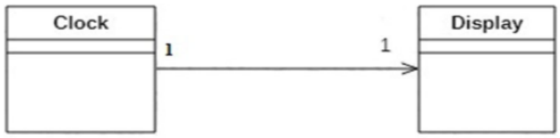
(c) 0..*

(d) 1..*

Q4. Consider the statement "room has chair" Which of the following types of association exist between room and chair?

(a) inheritance
(b) composition
(c) There is no association
(d) Aggregation

Q5. Consider the one-to-one unidirectional association exists in two classes: Clock Implementation and Display, as shown in figure.

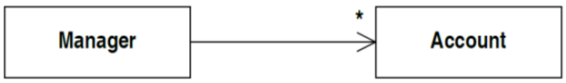


The correct implementation of the given figure is:

(a)	(b)

public class Clock {
private Display theDisplay;
……..
……..
}
	public class Display {
	private Clock theClock;
	……..
	……..
	}

Q6. Consider the one-to-many unidirectional association exists in two classes: Account and Manager, as shown in figure.



The correct implementation of the given figure is:

(a)	(b)

public class Manager {
private Account accounts[];
……..
……..
}
	public class Account {
	private Manager theManager;
	……..
	……..
	}

Q7. Consider the following example; two classes: Person and Account have a One-to-Many bi-directional association relationship between them, which means that one Person has many Accounts as shown in figure.

The correct implementation of the given figure is:

(a)	(b)

public class Persons {
private Account accounts;
……..
……..
}
	public class Account {
	private Person theOwner;
	……..
	……..
	}

Q8. Below are few real-world examples to understand the associations between them. Define type of association whether the aggregation or the composition.

- Association between Mobile store and mobiles
- Association between Car and Engine as mandatary part
- Association between a Bank and Employees
- Association between Library and Books
- Association between Institute, student and department
- Association between Building and rooms
- Association between Cities and State
- Association between Band and Musician

Q9. Let's take two classes, Professor class, and Department class. Below are the types of relationships/associations that can be possible between them. Define the forms of association whether one-to-one, one-to-many or many-to-many.

- One professor can only be assigned to work in one department.
- One professor can be assigned to work in multiple departments.
- Multiple professors can be assigned to work in one department.
- Multiple professors can be assigned to work in multiple departments.

Q10. Explain the unidirectional relationship? Describe how the one-to-one associations can be implemented.

Q11.What is an optional association? Explain the implementation of optional association with an example.

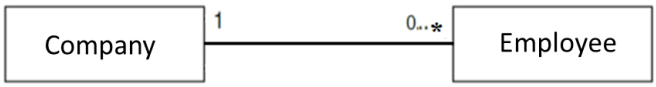
Q12. Explain one-to-one uidirectional association with an example in favour of implementation.

Q13. What is bi-directional relationship? How the bi-directional Implementations are made.

Q14. What is the difference between unidirectional and bidirectional association? Explain an approach to implementing bidirectional association.

Q15. How is a one-to-one bi-directional association different from one-to-many bi-directional association? Explain how one-to-one bi-directional association is implemented with the help of an example.

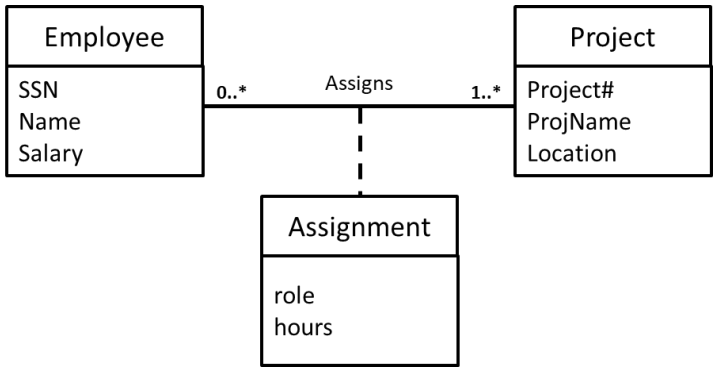
Q16. Consider the association relationship between Employee and Company class. How can we reduce the multiplicy using the qualified association?



(a) Draw UML diagram for reducing the multiplicy using the qualified association.
(b) Design the qualified association implementation using java code.

Q17. The many-to-many association between Employee and Project class that employees are assigned in one or more project and each project has zero or more employees assigned. The many-to-many association is replaced by a reified class – Assignment that maintains the role and hours of work in assigned project.

(a) Draw UML diagram with a reifed class (or) associate class
(b) Design the associate class implementation using java code.



Q16. Aggregation and Composition relationships are two forms of association. Consider the Car and Engine classes where two forms of association relationships as below:

- In the case of composition, the Car owns the Engine object (i.e. Engine cannot exist without the Car object).
- In the case of aggregation, the the Engine object is independent of the Car object (i.e. Engine can exist without the Car object).

(a) Draw UML diagrams for both forms of association between Car and Enginee class.
(b) Design two forms of association implementation using java code.

4. Implementing Constraints

A constraint describes a **condition** or an **integrity rule**. Constraints are typically used to describe the **permissible set of values of an attribute**,

- to specify the **pre- and post-conditions** for operations,
- to define certain **ordering of items**, etc.

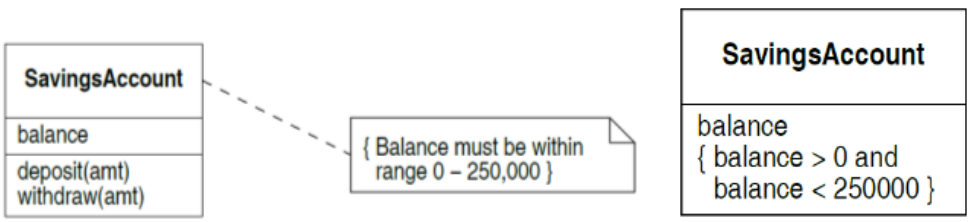
Constraints can be used to specify certain desirable properties of a class. **A class invariant** is **a property of a class** that is intended to be **true at all times for all instances of the class**. Class invariants describe relationships that must exist between the attribute values of an instance of the class, and preconditions and post conditions specify what must be true before and after an operation is called. Often the robustness of an implementation can be increased by including code in the class that checks these conditions at the appropriate times.

4.1 UML Notation for Constraints

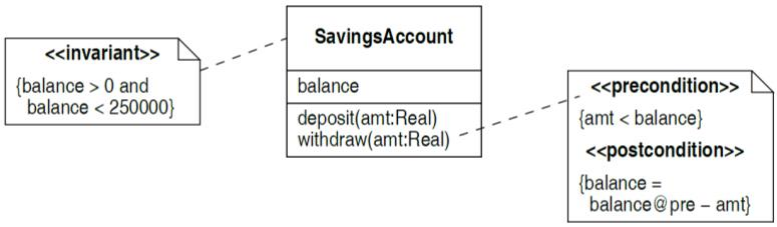
Constraint could have an optional name, though usually it is anonymous. A constraint is shown as a text string in curly braces according to the following syntax:

constraint ::= '{' [name ':'] boolean-expression '}'

For example, a bank is introducing a new type of savings account, which pays a preferential rate of interest. However, the balance of the account must remain within the range £0 to £250,000 and a deposit or withdrawal that would take the balance outside this range will be rejected. A class representing savings accounts is shown in Figure 1.35(a), with the constraint represented informally in a note and if the context is a class, the constraint can be placed inside the class icon shown in Figure 1.35(b), the account class with a constraint added to specify that an account's balance must fall within the specified limits. Figure 1.35(c) is stereotyped constraints with pre and post conditions in a note.



(a) constraint defined in a note (b) constraint added in attributes of a class



(c) Class specification using stereotyped constraints

Figure 1.35: A constrained savings account

4.2 Implementation Strategy for Constraints

The following example provides a possible implementation of the withdraw operation of the savings account class specified. A post condition was also given for this operation, specifying that the result of the operation was to deduct the amount given from the balance of the account. It might seem logical to check at the end of the operation that the post condition is satisfied. In general, preconditions are checked at the beginning of functions, and the class invariant can be checked at the end, as a check on the correctness of the implementation of the function.

The SavingsAccount Class

```
/*  The SavingsAccount class implementating the withdraw operation with precondition
 * constraints and post condition constraints  */
public class SavingsAccount
{
 private  String   accno;
 private  double  balance ;

 public SavingsAccount(String  ano) {
        accno  =  ano;
        balance = 0 ;
 }
 public boolean invariant() {
       return   balance >= 0 && balance <= 250000 ;
 }
// Precodition Constraints
 public void deposit(double amt)
         throws InvariantFailed, PreconditionUnsatisfied  {
        if (balance + amt > 250000) {
               throw PreconditionUnsatisfied ;
        }
        balance += amt ;
        if (!invariant()) {
               throw InvariantFailed ;
```

```
            }
    }
   //Post Condition Constraints
    public void withdraw(double amt)
                throws InvariantFailed, PreconditionUnsatisfied
    {
            if (amt > balance) {
                    throw PreconditionUnsatisfied ;
            }
            balance -= amt ;
            if (!invariant()) {
                    throw InvariantFailed ;
            }
    }
}
```

The PreconditionUnsatisfied Exception Class

```
class PreconditionUnsatisfied extends Exception {

 public PreconditionUnsatisfied()
 { super(); }

 public PreconditionUnsatisfied(String s)
 { super(s); }

}
```

The InvariantFailed Exception Class

```
class  InvariantFailed  extends  Exception {

 public InvariantFailed()
 { super(); }

 public InvariantFailed(String s)
 { super(s); }

}
```

The Test Deriver Class

```
 public class TestConstraints {
  //testing  constraints  for pre and post conditions for  a SavingsAccount.
```

```
public static void main(String[] args) {
      //Create Bank object
      SavingsAccount   acc = new  SavingaAccount("Acc001" );

      //deposit the balance
      acc.deposite(50000);
      // will throws InvariantFailed, PreconditionUnsatisfied
      acc.deposite(260000.00);

      //withdraw the balance
      acc.withdraw(250000.00);
         // will throw InvariantFailed because balance = 10000
}
```

Check Your Understanding on Implementing Constraints

Q1. What is a constraint? How do you show constraints on a class diagram?

Q2. Consider the Bank account **constraints** shown in the class diagram using a **note** symbol. (a) Draw another way of showing constraint in attributes. (b) Design the implementation in java code for it.

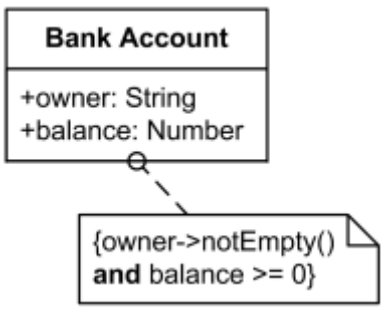
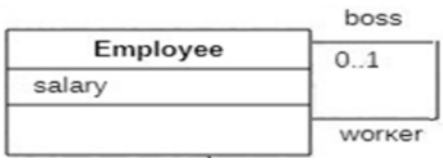


Figure Q2: Bank account constraints - snon empty owner and positive balance.

Q3. Draw a UML constraint diagram for Employee who could be as a boss or the worker. Worker's salary cannot exceed the salary of boss.



5. Implementing Statecharts Diagram

A state chart diagram describes how the **state of an object changes in its life time**. State chart diagrams are good at describing how the **behavior of an object changes across several usecase executions**. However, if we are interested in modeling some behavior that involves ***several objects collaborating with each other, state chart diagram is not appropriate***. We have already seen that such behavior is better modeled using sequence or collaboration diagrams. The state chart was proposed by David Harel [1990].

Following are the main purposes of using Statechart diagrams −

- To model the dynamic aspect of a system.
- To model the life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model the states of an object.

A state chart is **a hierarchical model of a system** and introduces the **concept of a composite state** (also called **nested state**). In start chart diagram

- **Actions** are associated with ***transitions*** and are considered to be processes that occur quickly and are not interruptible.
- **Activities** are associated with ***states*** and can take longer. An activity can be interrupted by an event.

5.1 The State Chart UML Diagram

The basic elements of the state chart diagram are as follows:

- **Initial state**: This represented as **a filled circle**.
- **Final state**: This is represented by **a filled circle inside a larger circle**.
- **State**: These are represented by **rectangles with rounded corners**.
- **Transition**: A transition is shown as **an arrow between two states**. Normally, the name of the event which causes the transition is places alongside the arrow.

A **guard** is a **Boolean logic condition**. The transition can take place only if the guard evaluates to true. The syntax for the label of the transition is shown in figure 1.36 has 3 parts— [guard]event/action.

An example state chart for the savings account object of the Bank software is shown in Figure 1.36. The statechart models **two states of a bank savings account**, recording whether it is in **credit** or **overdrawn**. The example assumes that the operations of deposit and withdraw money to and from the account, and guard conditions and actions document the effect of these operations in terms of the amount of money involved in the transaction, ‘amt’, and the current balance of the account, ‘bal’. Notice that when the account is overdrawn, no withdrawals can be made.

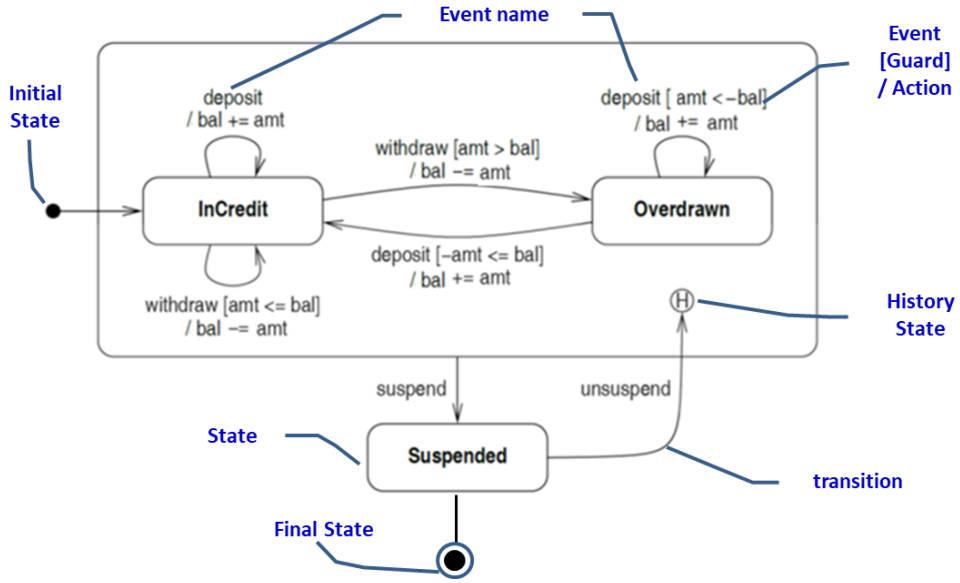


Figure 1.36: State chart diagram for a bank saving account object.

5.2 Implementation Strategies for the Statecharts Diagram

If a statechart is defined for a class, the information it contains can be used to structure the implementation of the methods of the class in a fairly mechanical way. This ensures that the behaviour of the class is consistent with that specified in the statechart, and in many cases this can make the job of implementing the class correctly much easier. This section summarizes a simple approach to implementing some features of a statechart. In Table 1.xx shows how to map the start chart elements to program constructs.

Table 1.xx: Mapping State machine elements to program constructs.

State Machine Element	Program Construct
State	State Class
Transition	method in StateMachine class
Event	Events class
Entry / Exit Actions	method in State class
Internal Action	method in State class
Hierarchical States	hierarchy of State Classes
Concurrent Transitions	method in the OrthogonalProperty class
Shallow History	attribute in StateMachine class
Deep History	History State Class

In above Figure 1.36, the statechart models two states of a bank account, recording whether it is in credit or overdrawn. The most important aspects of **any implementation of the behaviour specified in a statechart** are to **record the current state of an object** and to ensure that its **response to a message is correctly determined by its current state**.

The approach for implementing statecharts is using the **switch statement**. The **states of statecharts diagrams** are denoted as **objects** or **scalar variables**. **Events** and **actions** are indicated as **methods**. The most basic technique to record the current state is **to enumerate the different states as constants** and **to store the current state in a suitable datamember**. The following code shows how this could be done, and also shows how **the constructor sets the initial state of the object** to '***InCredit'***, as specified in Figure 1.36.

```
public class Account
{
        private  final  int  InCredit = 0 ;      // state variable
        private  final  int  Overdrawn = 1 ;  // state variable
        private  final  int  Suspended = 2 ;  // state variable
        private  int  state ;

        public Account() {
             state = InCredit ;    //initial state of an account object
        }
}
```

(1) Operations whose effect is **state-dependent can be implemented** as **switch statements**, with **a separate case for each state**.

- **Each case represents all the transitions leading from the specified state** that are ***labelled with the appropriate message***.
- It should check any applicable **guard conditions**, perform any **actions** and if necessary **change the state of the object** by ***assigning a new value*** to the data member recording the state.
- If an operation is **not applicable in a given state**, that case can simply be ***left empty***.

For example, the following is a possible implementation of the '**withdraw' operation** in the **account** class.

```
public void withdraw(double amt) {
    switch (state) {
    case  InCredit:
                    if (amt > balance) {
                        state = Overdrawn ;
                    }
```

```
                balance -= amt ;
                break ;
        case Overdrawn:
        case Suspended:
        break ;
        }
    }
```

(2) Composite states are essentially a device for simplifying the structure of a statechart and do not need to be represented as separate states. Their major role is to denote by a single transition a set of equivalent transitions from each of their substates. The '**suspend' transition** in Figure 13.14 provides an example of this. This transition can be implemented simply by **grouping together the cases that refer to the substates**, as shown in the following outline implementation of **the 'suspend' operation in the account class**.

```
public void suspend() {
    switch (state) {
        case InCredit:
        case Overdrawn:        //group all substates as one case
            state = Suspended ;
            break ;
        case Suspended:
            break ;
    }
}
```

(3) History states are ***not additional states in a statechart***, but rather **a device for recording the most recent substate in a composite state**. The history state in Figure 13.14 can therefore be **represented by a variable storing a state**, and this stored state can be used when an account is unsuspended, as shown below.

```
private int historyState ;
public void unsuspend() {
    switch (state) {
        case Suspended:            //group all substates as one case
            state = historyState ;
            break ;
        // other cases
    }
}
```

Two bits of housekeeping need to be performed on the history state. The first time the composite state is entered, the history state is not active and the account enters the

'InCredit' state, as specified by the initial state in the composite state. In general, this behaviour can be simulated by **initializing the history state variable to the specified initial state.**

In addition, the history state variable needs to be **set whenever a transition leaves the composite state**. In the current example, this would be in the suspend method, just before the state is set to Suspended.

This general approach to the implementation of statecharts is simple and generally applicable, but it also has a few disadvantages. First, it **does not provide much flexibility in the case where a newstate is added to the statechart**. In principle, the implementation of every member function of the class would have to be updated in such a case, even if they were quite unaffected by the change. Second, the strategy assumes that most functions have some effect in the majority of the states of the object. If this is not the case, **the switch statements will be full of 'empty cases' and the implementation will contain a lot of redundant code**.

The Account Class

```
public class SavingsAccount
{
    private final int InCredit = 0 ;      // the state variable of an account object
    private final int Overdrawn = 1 ; // the state variable of an account object
    private final int Suspended = 2 ; // the state variable of an account object
    private int historyState ;       // the state variable of an account object
    private   int   state ;
    private String   accno;
    private double  balance ;

  public   SavingsAccount() {
       state = InCredit ;     // define the initial state of an account object
       accno  =  ano;
       balance = 0 ;
  }

// Events and actions are indicated as methods
 public void deposit(double amt)
 {
       switch (state) {
       case  InCredit:
                balance  + =  amt;
                break ;
       case  Overdrawn:
                if ( amt  < - balance ) {
                   state = Overdrawn ;
```

```
                }
                If ( - amt <= balance ){
                    state = InCredit ;
                }
                balance  + =  amt;
                break ;
        case  Suspended:
                suspend();
                break ;

 }

// Events and actions are indicated as methods
 public void withdraw(double amt) {
        switch (state) {
        case  InCredit:
                if (amt <= bal) {
                      state = InCredit ;
                }
                if (amt > bal) {
                      state = Overdrawn ;
                }
                balance -= amt ;
                break ;
        case  Overdrawn:
        case  Suspended:
                suspend();
                break ;
        }
    }

    public void suspend() {
        switch (state) {
            case  InCredit: case  Overdrawn:
              //Current state to be restored when leaving the composite state
                  historyState  =  state;
                  state  = Suspended;
                  break;
            case  Suspended:
                   break ;
        }
    }

    public void unsuspend() {
        switch (state) {
```

```
                case Suspended:
                  //Last state to be reinstated when unsuspend is triggered
                    state = historyState ;
                    break ;
                // other cases
            }
        }
    }
```

Check Your Understanding on Implementing Statecharts Diagram

Q1. A state chart is composed of all of the following except _____.

a. a set of states
b. a set of possible input events
c. a function that maps current state and input to next state
d. an instruction set

Q2. Consider the following statechart diagram shown in figure for the video player application with two states: Stop and Play. In the Stop state, when a user presses a play button, then playButton event happens, and the device executes the Playstart action and goes to 'Play' state. In the Play state, when the user presses a stop button, the stopButton event generates, and the device executes the Playstop action and goes to the Stop state. Design implementation for the statechart as shown in figure.

Figure : Statechart for a video player

6. Implementing the Interaction Diagrams

The interaction diagrams in UML are the sequence and collaboration diagram. They depict the relationships between the objects in a system and objects work together to perform a particular task. It **defines a functionality of the system** by demonstrating **a set of chronological interactions between objects**. It is basically used to demonstrate how objects interrelate to **perform the behaviour of a specific use case**.

6.1 Creating Methods from the Collaboration Diagram

A collaboration diagram carries the similar type of information as a sequence diagram; it focuses on the ***object structure instead of the chronology of messages passing between them***. The collaboration diagram is also called **a communication diagram** and is used to **represent the object's architecture in the system**. Collaboration diagrams are appropriate for **analyzing use case diagrams**. This diagram can be used to show the **dynamic behaviour of a specific use case** and describes the **role of each object.**

The key purpose of creating the collaboration diagram is

- to illustrate the **relationship between objects** and works together to perform a particular task. This task is difficult to regulate from a sequence diagram.
- to support developer in controlling the **accuracy of the static model**, such as **class diagrams**.

In an application system, there are one or more collaboration diagrams to find all the eventuality of a complex behaviour. The collaboration diagram contains the elements such as **actors**, **objects** and their **communication links**, and **messages**.

- **Each actor** performs a significant role as it **invokes the interaction and has a name**.
- The **link** behaves as **a connector between the objects and actors**. It depicts a relationship between the objects through which the messages are sent.
- The **messages** are **exchanged between objects in an order** which is represented by sequence numbers. The messages are sent from the sender to the recipient, and the direction must be traversable in that particular direction.

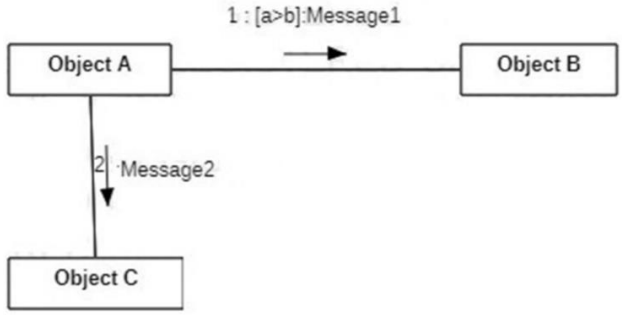


Figure 1.37: Collaboration Diagram showing If-else condition

The implementation of the above diagram, Figure 1.37 is as follows:

```
if (a>b) {
        ObjectB.Message1();
}
else {
        ObjectC.Message2();
 }
```

6.2 Implementing the Collaboration Diagram Example

Let us consider a collaboration diagram for ATM transactions. For this example, first, create a Use Case diagram and then draw a collaboration diagram. For building **a Use Case diagram**, use the below flow of controls:

- Customer can insert ATM card into the card reader of the ATM console.
- The card reader reads and verifies the card.
- ATM console displays a message and prompts the customer to enter PIN.
- Customer enters PIN.
- ATM console verifies PIN with bank database and confirms the PIN is valid.
- If PIN is invalid, ATM console notifies the customer with a message and ejects the card.
- If PIN is valid, ATM console represents a prompt with options such as deposit, withdraw and transfer money.
- Customer selects withdraw option.
- ATM console presents prompt to the customer for an accurate amount to be withdrawn.
- The customer enters the amount.
- ATM console verifies with the bank database whether the customer account has sufficient funds. Otherwise, it shows an insufficient funds message.
- ATM console deducts money from customer's account, puts the requested amount in cash dispenser, and prints a receipt.
- ATM console rejects the card and becomes ready for another transaction.

Using the above flow of messages, construct the collaboration diagram similar to figure 1.38 where messages/method calls are flowing between objects with a sequence number. This sequence number specifies the ordering of the messages. The messages appear with pointing arrows from the sender object to the receiver object.

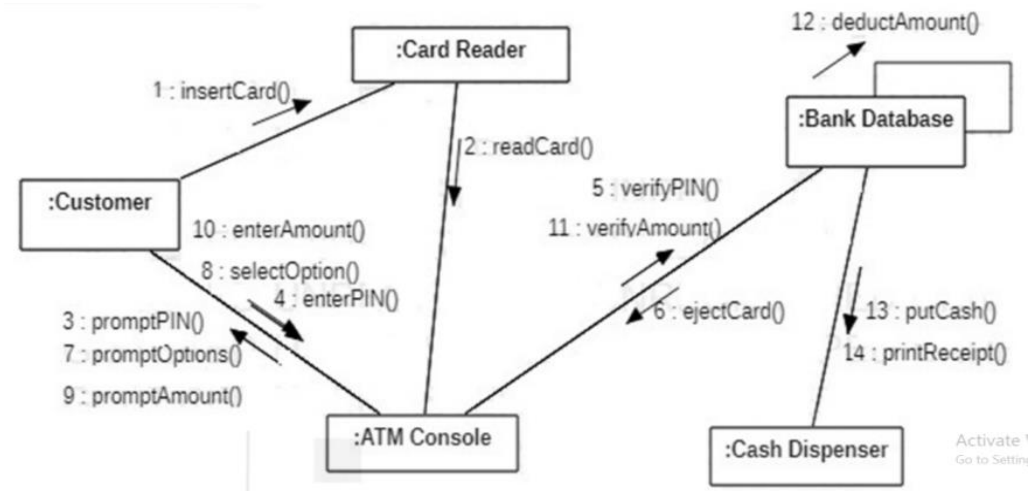


Figure 1.38: Collaboration Diagram for ATM transaction

When creating a collaboration diagram for the real business application, we can create methods as per need. The classes and their methods are created in the Collaboration Diagram given in figure 1.38, which can be summarized in the tabular form:

Table 13.1: Description of Classes and Methods

Class Name	Methods	Class Name	Methods
Customer	insertCard() enterAmount() selectOption() enterPIN()	Card Reader	readCard()
ATM Console	promptPIN() verifyPIN() promptOptions() promptAmount() verifyAmount()	Bank Database	ejectCard() deductAmount() putCash() printReceipt()

Now you can transform methods defined in the collaboration diagram into executable code. There are number of methods in the figure-1.38, but the pseudocode given below is written for verifyPIN() method only. This method verifies the customer's Personal Identification Number (PIN) using data retrieved from the ATM card's magnetic strip.

This method asks the customers to enter their PIN using the keypad of ATM console. If the PIN does not equal to the PIN stored on the card, then the ATM system allows a limited number of repeats. If it is still not matched, then the system invokes ejectCard() method and the card is seized as a security precaution. If the customer enters the correct PIN then the system invokes promptOptions() method where it provides options such as deposit money, withdraw cash and transfer money. You may try to write code for other methods as well.

The Pseudo Code for verifyPIN method as follows:

```
method verifyPIN
    constants no_of_maxpins is 2
    variable pin, pin_count is number
    set pin_count to zero
    read pin from ATMcard
    loop until pin_count is equal to no_of_maxpins
            input pin from customer
            if entered pin matches with cardpin_database then
                call promptOptions method
            endif
            add 1 to pin_count
    endloop
    if pin_count is equal to no_of_maxpins then
       seize customer's card
    else
       call ejectCard method
    endif
endmethod// verifyPIN
```

The Java Code for verifyPIN method as follows:

```
public  boolean verifyPIN(ATMCard  ATMcard) {
     int constants no_of_maxpins = 2;
     int cardpin, pin_count. enteredpin;

     pin_count= 0;
     cardpin = ATMcard.readpin();  // pin stored in card
     while (pin_count == no_of_maxpins ){
          //pin entered by customer
          System.out.print("Enter pin: " ); input(enteredpin);
          if (enteredpin  == cardpin) then
               promptOptions();
          }
```

```
            pin_count++;
        }//endloop
        if (pin_count == no_of_maxpins) {
            seizecustomercard();
        else
            ejectCard();
        } //endif
    }// endmethod of verifyPIN
```

6.3 Implementing the Collaboration Diagram Example 2

Consider another example of **'Programme Management System'**. Using this example, you can learn about the creation of methods from the collaboration diagram. This system contains **classes** like **Admin, ProgrammeCoordinator, Programme, Course and Faculty.**

- **Admin** initiates the manage programme functionality and interacts with Programme Coordinator.
- The **ProgrammeCoordinator** intermingled with different object classes in the system, such as Programme, Course and Faculty. The Programme Coordinator starts the process of creation and modification functionality of the programme.
- After the **programme** is either created or modified, ProgrammCoordinator is responsible for the creation or modification functionality of a course.
- Each **programme** consists of many courses. It means that the class Programme and class Course are associated with each other in the form of one-tomany relationships.
- The **ProgrammeCoordinator** assigns the course(s) to faculty.
- Finally, the admin invokes the assigned faculty to the Course functionality, of the Programme.

The Collaboration Diagram for Programme Management System is shown in figure 1.39:

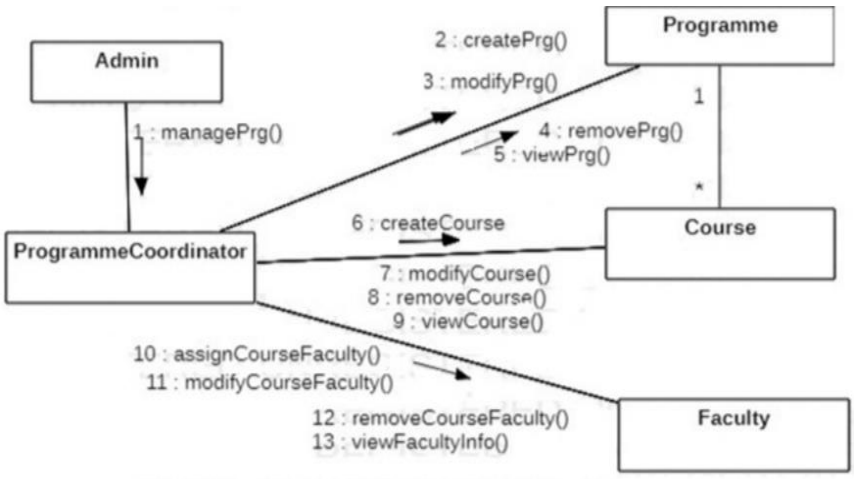


Figure 1.39: Collaboration Diagram for Programme Management System

The collaboration diagram for 'Programme Management System' shown in figure 1.39, involves the classes and contains the following methods. You may write code for the methods given in figure 1.39. List the basic assumptions, then try to write the code in the programming language you are comfortable.

Table 13.2: Description of Classes and Methods in Collaboration Diagram

Class Name	Methods	Class Name	Methods
Admin	managePrg()	Faculty	assignCourseFaculty() modifyCourseFaculty() removeCourseFaculty() viewFacultyInfo()
Programme	createPrg() modifyPrg() removePrg() viewPrg()	Course	createCourse() modifyCourse() removeCourse() viewCourse()

6.4 Implementing the Sequence Diagram

Consider the stock control program that stores details about orders which consist simply of a number of order lines, each of which specifies a certain number of parts of a particular type. Figure 1.40 shows a class diagram summarizing the necessary facts about orders. The relationship between orders and order lines is one of composition, because assume that when an order is destroyed all lines on that order are also destroyed.

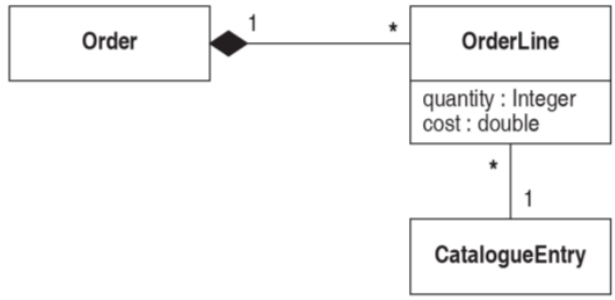


Figure 1.40: Orders in the stock control system

Let us assume that to create an order line, a client object must send a message to an order specifying the part and the quantity to be ordered. A new order line object will then be created and added to the order. A sequence diagram illustrating this interaction is shown in Figure 1.41.

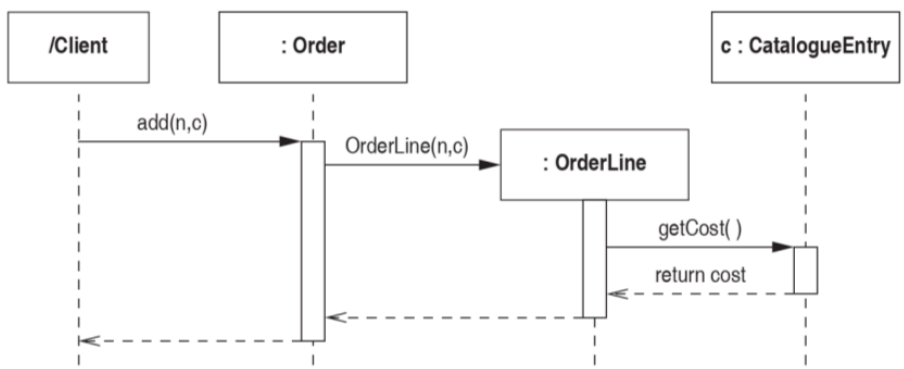


Figure 1.41: Sequence diagram for creating an order line

The new order line object is created in response to a message labelled ‘add’ sent from the client to the order. The parameters of this message supply the number of parts to be added and the catalogue entry object corresponding to the required parts. In response to this message, an order object creates a new order line object and the constructor call creating this object is shown as the next message in the interaction. The diagram shows that during the construction of order lines the cost of the relevant parts is retrieved from the catalogue entry object that was passed as a parameter. This value will be used in the initialization of the order line’s ‘cost’ attribute. At the end of this activation, control returns to the order and the order line’s lifeline continues as normal.

Collaboration diagrams cannot explicitly show the time at which a new object is created. In order to distinguish elements that are created during the course of an interaction from those that existed at the start, classifier and association roles

corresponding to new objects and links are annotated with the property 'new', shown in Figure 1.42.

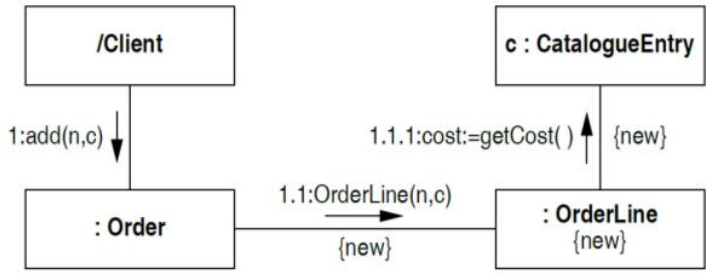


Figure 1.42: Object creation on a collaboration diagram

The classes and their methods are created in the Sequence Diagram given in figure 1.41, which can be summarized in the tabular form:

Table 13.1: Description of Classes and Methods

Class Name	Methods	Class Name	Methods
Order	OrderLine(int n, Catalogue c)	CatalogueEntry	
OrderLine	getCost()	ClientInterface	add(int n, Catalogue c)

Now transform interaction process defined in the Sequence Diagram into executable code. However only involved methods in the Sequence Diagram are defined as follows:

The Order.java, OrderLine.java, CatalogueEntry.java, Part.java

```
public class  Order {

    private  List<OrderLine>  lineItems  = new ArrayList<OrderLine>();

    …..  //other mehods

    //create order line object and  add into ArrayList
    public boolean OrderLine(int n, Catalogue c) {
        lineItems.add(new OrderLine(n, c));
        return true;
    }
    ….  // other methods
    public  void  toString(){
        System.out.println( "Order includes: ");
        for(OrderLine  item : lineItems){
            System.out.println( item.getCatName(); + " , "  +  item.getQty() +  " , " +
```

```
                                        lineItem. getCost());
    }
}
public class  OrderLine {

    private  int  qty;
    private  CatalogueEntry  catalog;
    private  double  cost;

    public  OrderLine (int n, CatalogueEntry c) {
        this.qty = n;
        this.catalog  = c;
        this.cost =  c.getCost();
    }

    public  String  getCatName() { return  catalog.getName() ; }
    public  int  getQty() { return  qty ; }
    public  double  getCost() { return cost ; }
}
public class CatalogueEntry
{
    private String name ;
    private long number ;
    private double cost ;

    public CatalogueEntry(String nm, long num, double cst) {
        name = nm ;
        number = num ;
        cost = cst ;
    }
    public String getName() { return name ; }
    public long getNumber() { return number ; }
    public double getCost() { return cost ; }
}
```

The test driver TestOrder.java

```
public class  TestOrder {
    public static void main(String[] arg){
        CatalogueEntry  screw = new CatalogueEntry("screw", 28834, 0.02) ;
        CatalogueEntry  bolt = new CatalogueEntry("bolt", 10117, 0.03) ;
        CatalogueEntry  hammer = new CatalogueEntry("hammer", 10223, 10.00) ;
        CatalogueEntry  nail = new CatalogueEntry("nail", 10118, 0.04) ;

        Order  order1 = new Order();
```

```
        order1.add(3, screw);
        order1.add(4, bolt);
        order1.hammer(1, hammer);
        System.out.println(order1.toString());

        Order order2 = new Order();
        order2.add(20, nail);
        order2.add(10, bolt);
        System.out.println(order2.toString());
    }
```

Example 2:

Consider the book issued by the librarian when customer requests to issue a book. The librarian first check the book is available. If yes, validate the member and checked the number of books issued by the member. If book can be issued, create issued transaction for the book and add member and book details. Then update the book status and member issued books status. The Collaboration diagram and Sequence diagram are shown below.

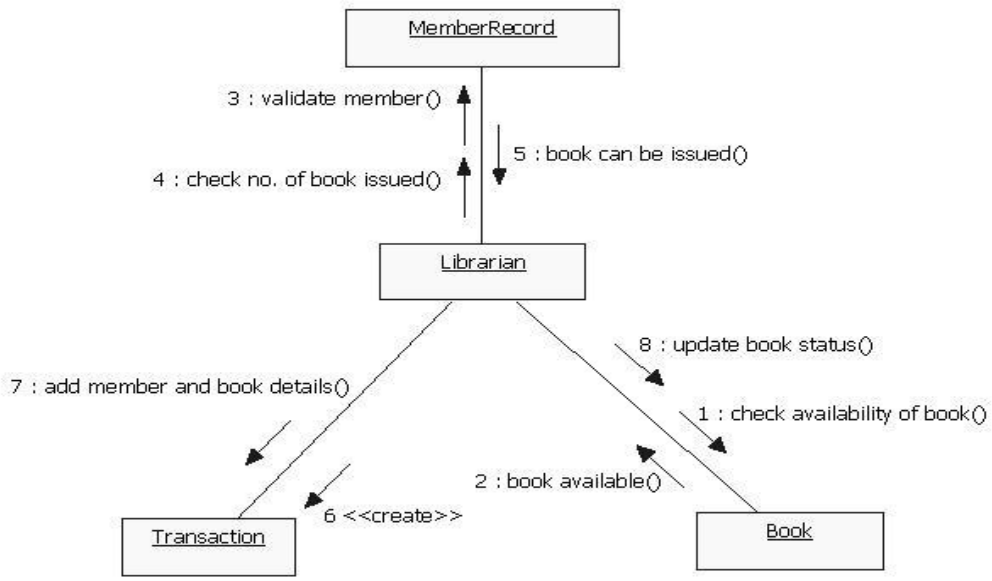


Figure 1.43: Collaboration diagram for issuing Book

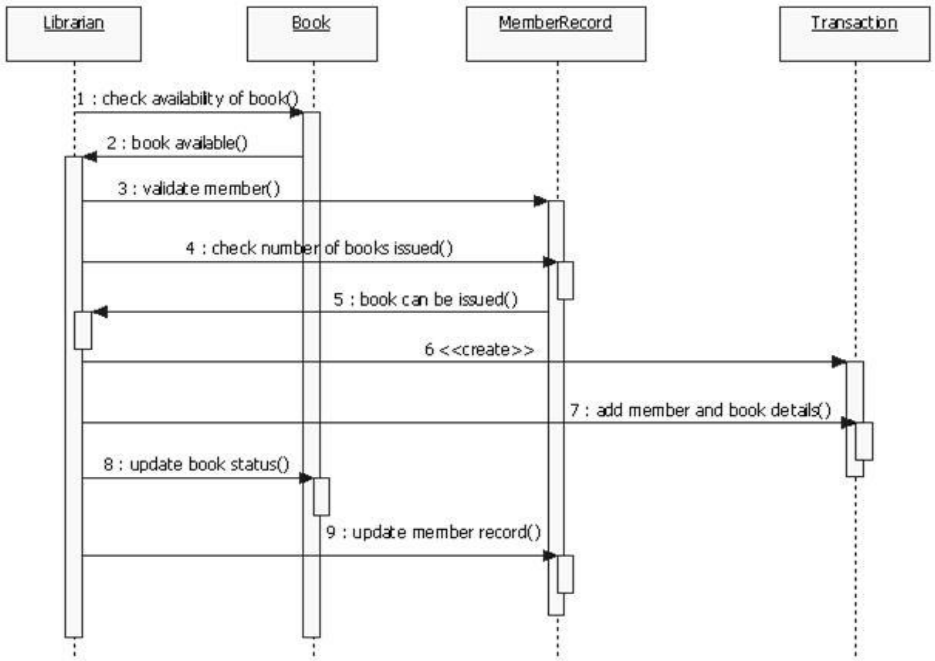


Figure 1.44: Sequence diagram for issuing Book

The classes and their methods are created in the Sequence Diagram given in figure 1.44, which can be summarized in the tabular form:

Table 13.1: Description of Classes and Methods

Class Name	Methods	Class Name	Methods
Librarian	checkAvailableofBook() updateBookStatus() validateMember() checknoofbooksIssued() addmemberandbookDetails() updateBookStatus() updatememberrecord()	MemberRecord	BookcanbeIssued()
Book	BookAvailable()	Transaction	

Now transform interaction process defined in the Sequence Diagram into executable code. However only involved methods in the Sequence Diagram are defined as follows:

The Librarian.java, Book.java, Transaction.java, MemeberRecord.java

```
public class  Librarian {
```

```
    BookCatalogue   catalog = new  BookCatalogue();
    MemberList    memberList  =  new MemberList();

    …..  //other mehods

    public boolean IssuedBook(int   bookID, int   memberID) {
        //get book object from book catalogue List
        Book  book = catalog.search(bookID);
        if   (book.checkBookAvailable()) {
            System.out.println( "Book is available to issue." );
        } else {
            System.out.println( "Book is not available" );
            return false;
        }

        //get member object from  memberList
        MemberRecord   member  =  memberList.validateMember(memberID);
        if (member == null ) {
            System.out.println( "Invalid Member!" );
            return false;
        }  else {
            System.out.println( "Member is  valid." );
        }
        if (member.checkBookIssued()) {
            System.out.println( "Number of Books issued is under the Limit." );
        } else {
            System.out.println( "Number of Books issued is over the Limit." );
            return false;
         }
        // create issue book transaction
        Transaction   trans  =  new  Tranction( );
        trans.addIssuedBook(bookID(),  memberID);
        System.out.println( "Book is issued successfully." );

        //update
        book.updateBookStatus(false);
        member.updateMemberRecord(bookID);
        return true;
    }
    ….  // other methods
}
```

```
public class  Book {

    private  int  bookID;
    private  String  bookTitle;
```

```
    private  String  bookAuthor;
    private  String  bookPublisher;
    private  boolean  bookAvail = true;

    …..   // other methods
    public boolean checkBookAvailable(){
         return  this.bookAvail;
    }
    public void updateBookStatus(boolean  status){
        this.bookAvail = status;
    }
    public  int  getID(){    return  bookID;   }
    public   String   getTitle() {    return  bookTitle;   }
    public  String   getAuthor(){     return   bookAuthor;   }
    private  String  getPublisher() {     return   bookPublisher;   }

    ….  // other methods
}
```

```
public class  MemberRecord {

    private  int  memberID;
    private  String  memName;
    private  int   bookLimit;
    private  List<int>   borrowedBooks  = new ArrayList<int>();

    …  //other methods
    //Check the member ID is valid member or not
    public boolean  validateMember(int  mID){
        if (memberID == mID)
           return true;
        else
           return  false;
    }
    //Check member borrowed books  exceed the limit
    public boolean  checkBookIssued( ){
        if (borrowedBooks.count() < bookLimit)
              return  true;
         else
              return  false;
    }
    //Record the borrowed book ID
    public void updateMemberRecord(int   bookID){
        borrowedBooks.add(bookID);
    }
```

```
}
public class Transaction {

    ptivate  int  transNo;
    pricate   date   transDate;
    private  int  memID;
    private  List<book>  issuedBook = new ArrayList<Book>;

    public Transaction( int  tno,  date  tdate){
        this.transNo  =  tno;
        this.transDate  =  tdate;
    }

    public  void  addIssuedBook(int bID, int  mID){
      this.memID  = mID;
      issuedBook.add(bID);
    }
    //…. other methods
}
```

Check Your Understanding on Implementing the Interaction Diagrams

Q1. The purpose of the interaction diagram is to

(a) Discover and define the communication between objects
(b) Discover and define the sequence of interactions between objects
(c) Discover and define the interfaces on classes
(d) Discover and define the attributes that define an object

Q2. What elements are discovered using the sequence diagram and added to the class diagram?

(a) Interfaces of the objects
(b) Classes of the objects in the sequence diagram, attributes, and operations
(c) Classes of the objects in the sequence diagram
(d) Operations and the associated attributes that appear as arguments in the operation definition

Q3. Where can you find attributes in a sequence diagram?

(a) In operation arguments
(b) In the objects that own the parameters' values
(c) In the responses to events
(d) In operation parameters and responses

Q4. How does the sending object obtain data from the object that owns the parameter values?

(a) By adding an interface to the sending object and adding it to the interaction sequence
(b) By adding an interface to the owning object and adding it to the interaction sequence
(c) By reassigning the attributes to the object that receives the parameters
(d) By reassigning the attributes to the object that sends the parameters

Q5. What are the types of UML interaction diagrams and how do they differ?

Q6. UML sequence diagrams typically show some sequence of method invocations that achieve some specific purpose. Figure Q1 shows a sequence diagram for calculating the total of a sale transaction. It starts with a call to the method calcTotal() of the Sale class. Design the relevant source code from the diagram shown.

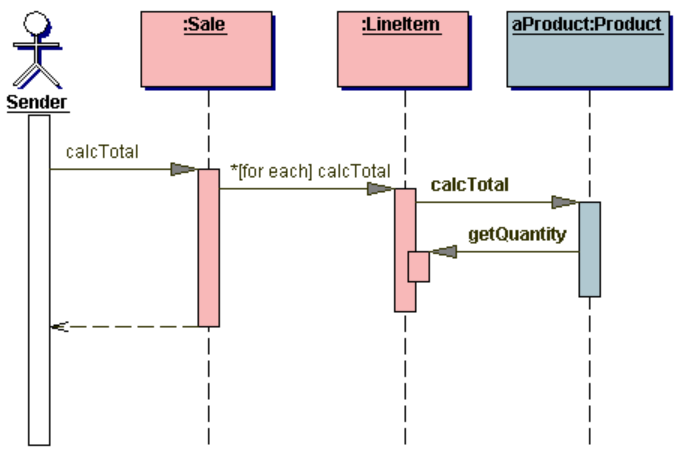


Figure Q6: A sequence diagram for calculating the total of a sale.

Q7. UML collaboration diagram describes the same interactions as the sequence diagrams of sequence diagram. Figure Q2 shows a collaboration diagram for students to register in a course. List the methods from the diagram and design the relevant source code for each methods.

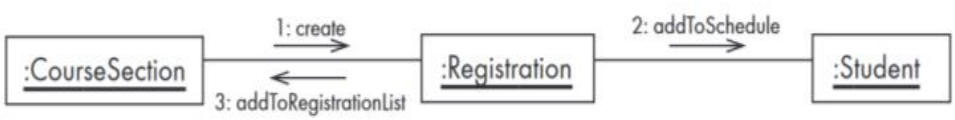


Figure Q7: A collaboration diagram showing the student registration process

7. SOLID: Single, Open-Closed, Liskove Substitution and Dependency Inversion Principles

The SOLID principles are the fundamental principles that suggest for the object-oriented programming that -

- these principles are used to **determine how we should design classes**.
- these principles are used to **design software applications maintainable and testable**.

SOLID are principles, not patterns. SOLID is an acronym for the **five design principles** introduced by Robert C. Martin. SOLID stands for:

(1) S = Single Responsibility Principle
(2) O = Open/Closed Principle
(3) L = Liskov Substitution Principle
(4) I = Interface Segregation Principle
(5) D = Dependency Inversion Principle

7.1 The Single Responsibility Principle (SRP)

The single responsibility principle states that:

"Each software module should have one and only one reason to change."

In other words, **a class should have only one responsibility** and therefore it should have only one reason to change its code. If a class has more than one responsibility, then there will be more than one reason to change the class (code).

Now, the question is **what is responsibility?**

An application can have much functionality (features). For example, an online e-commerce application has many features such as displaying product lists, submitting an order, displaying product ratings, managing customers' shipping addresses, managing payments, etc. Along with these features, it also validates and persists products and customers' data, logs activities for auditing and security purposes, applies business rules, etc. You can think of these points as **functionalities or features or responsibilities**. **Change in any functionality leads to change in the class** that is responsible for that functionality.

Let's check how many responsibilities the following `Student` class has:

Example: A Class with Multiple Responsibilities

```
//Class 1: Student Class
public class Student  {

   public int StudentId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public DateTime DoB { get; set; }
   public string email { get; set; }
   public string Address1 { get; set; }
   public string Address2 { get; set; }
   public string City { get; set; }
   public string State { get; set; }
   public string Zipcode { get; set; }

   public void Save()    {
      Console.WriteLine("Starting Save()");
      //use EF to save student to DB

      Console.WriteLine("End Save()");
   }
   public void Delete()    {
      Console.WriteLine("Starting Delete()");

      //check if already subscribed courses then don't delete

      Console.WriteLine("End Delete()");
   }
   public IList<Course> Subscribe(Course cs)    {
      Console.WriteLine("Starting Subscribe()");

      //apply business rules based on the course type
      if(cs.Type == "online")       {
         //validate
      }
      else if(cs.type == "live")      {

      }

      //payment processing code
         ……..
      //save course subscription to DB
         ……..
      //send email confirmation code
         ……..
      Console.WriteLine("End Subscribe()");
   }
}
```

The above `Student` class has the following **responsibilities**:

1. Holds student's properties such as `StudentId`, `FirstName`, `LastName`, and `DoB`.
2. Save a new student, or update an existing student to a database.
3. Delete existing students from the database if not subscribed to any course.
4. Apply business rules to subscribe to courses based on the course type.
5. Process the payment for the course.
6. Send confirmation email to a student upon successful registration.
7. Logs each activity to the console.

If anything in the **above responsibility changes**, then we will have to **modify the `Student` class**. For example,

- if you need to **add a new property** then we **need to change** the `Student` class.
- Or, if you need a **change in the database**, maybe moving from a local server to a cloud, then you need to **change the code** of the `Student` class.
- Or, if you need to **change the business rules** (validation) before deleting a student or subscribing to a course, or **change the logging medium** from console to file, then in all these cases you **need to change the code** of the `Student` class.

Thus, you have many reasons to change the code because it has many responsibilities.

7.1.1 Applying SRP

SRP tells us to have only one reason to change a class. Let's change the `Student` class considering SRP where we will **keep only one responsibility** for the `Student` class and abstract away (delegate) other responsibilities to other classes.

Start with each responsibility mentioned above and **decide whether we should delegate it to other classes or not**.

1. The `Student` class should contain all the properties and methods which are specific to the student. Except for the `Subscribe()` method, all the properties and methods are related to the student, so keep all the properties.
2. The `Save()` and `Delete()` method is also specific to a student. Although, it uses Entity Framework to do the CRUD operation which is another reason to change the Student class. We should move the underlying EF code to another class to do all DB operations e.g. StudentRepository class should be created for all CRUD operations for the Student. This way if any changes on the DB side then we may need to change only the StudentRepository class.

3. The `Subscribe()` method is more suitable for the Course class because a course can have different subscription rules based on the course type. So it is idle to move the `Subscribe()` method to the Course class.
4. Sending confirmation emails is also a part of the `Subscribe()` method, so it will be a part of the `Course` class now. Although, we will create a separate class `EmailManger` for sending emails.
5. Here, all the activities are logged on the console using the hard-coded `Console.WriteLine()` method. Any changes in the logging requirement would cause the `Student` class to change. For example, if the admin decides to log all activities in the text file then you need to change the `Student` class. So, it's better to create a separate `Logger` class that is responsible for all the logging activities.

Now, look at the following **classes redesigned after applying SRP** using the above considerations for SRP.

Example: Class After Applying SRP

```
// Class 1: Student class
public class Student
{
   public int StudentId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public DateTime DoB { get; set; }
   public string email { get; set; }
   public string Address1 { get; set; }
   public string Address2 { get; set; }
   public string City { get; set; }
   public string State { get; set; }
   public string Zipcode { get; set; }

   public void Save()    {
      Logger.Log("Starting Save()");

      _studentRepo.Save(this);

      Logger.Log("End Save()");
   }

   public void Delete()   {
      Logger.Log("Starting Delete()");

      //check if already subscribed courses

      _studentRepo.Delete(this);

      Logger.Log("End Delete()");
```

```
   }
}

// Class 2:  Logger Class
public class Logger
{
   public static void Log(string message)  {
      Console.WriteLine(message);
   }
}

// Class 3:  StudentRepository Class
public class StudentRepository()
{
    public bool Save(Student std)  {
      Logger.log("Starting Save()");

      //use EF to add a new student or update existing student to db

      Logger.log("Ending Saving()");
   }

   public bool Delete()   {
      Logger.log("Starting Delete()");

      //use EF to delete a student

      Logger.Log("Ending Delete()");
   }

   public bool SaveCourse(Student std, Course cs)  {
      Logger.log("Starting SaveCourse()");

      //use EF to save a course for a student

      Logger.Log("Ending SaveCourse()");
   }
}

// Class 4:  Course Class
public class Course  {
   public int CourseId { get; set; }
   public string Title { get; set; }
   public string Type { get; set; }

   public void Subscribe(Student std)  {
      Logger.Log("Starting Subscribe()");

      //apply business rules based on the course type live, online, offline, if any
      if (this.Type == "online")
      {
         //subscribe to online course
```

```
        }
        else if (this.Type == "live")
        {
            //subscribe to offline course
        }

        // payment processing
        PaymentManager.ProcessPayment();

        //create CourseRepository class to save student and course into StudentCourse table

        // send confirmation email
        EmailManager.SendEmail();

        Logger.Log("End Subscribe()");
    }
}

// Class 5:  EmailManager Class
public class EmailManager {
    public static void SendEmail(string recEmailed, string senderEmailId,
                                 string subject, string message)  {
            // smtp code here
    }
}

// Class 6:  PaymentManger  Class
public class PaymentManger  {
    public static void ProcessPayment()  {
            //payment processing code here
    }
}
```

Now, think about the above classes. **Each class has a single responsibility**.

- The `Student` class contains properties and methods specific to the student-related activities.
- The `Course` class has course-related responsibilities.
- The `StudentRepository` has responsibilities for student-related CRUD operations using Entity Framework.
- The `Logger` class is responsible for logging activity.
- The `EmailManager` class has email-related responsibilities.
- The `PaymentManager` class has payment-related activities.

In this way, we have **delegated specific responsibilities to separate classes** so that each **class has only one reason to change**. These **increase cohesion and loose coupling**.

7.1.2 Separation of Concerns

The Single Responsibility Principle (SRP) follows another principle called Separation of Concerns.

Separation of Concerns suggests that the application should be **separated into distinct sections where each section addresses a separate concern** or set of information that affects the program. It means that high-level business logic should avoid dealing with low-level implementation.

In our example, we separated each concern into separate classes. We had only one `Student` class initially, but then we separated each concern like CRUD operations, logging, email, etc. into separate classes. Thus, the `Student` class (high-level class) does not have any idea how CRUD or sending emails is happening. It just uses the appropriate method and that's it.

The **SRP and Separation of Concerns principle increase cohesion and loose coupling**.

7.2 Open-Closed Principle

The Open/Closed Principle (OCP) is the second principle of SOLID. Dr. Bertrand Meyer originated this term in his book Object-oriented Software Construction. Open/Cloced Principle states that:

"**Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification**."

Now, what does it mean by **extension and modification**?

Here, the **extension means adding new features to the system without modifying that system**. The **plugin systems** are the main example of OCP where new features are added using new features without modifying the existing ones.

OCP says the behavior of a method of a class should be changed without modifying its source code. You should not edit the code of a method (bug fixes are ok) instead you should use polymorphism or other techniques to change what it does. Adding new functionality by writing new code.

In Java and C#, Open/Closed principle can be applied using the following approaches:

1. Using Function Parameters
2. Using Extension methods
3. Using Classes, Abstract class, or Interface-based Inheritance
4. Generics
5. Composition and Dependency Injection

To demonstrate OCP, let's take an example of the `Logger` class shown below. Assume that you are the creator of this class and other programmers want to reuse your class so that they don't have to spend time to rewriting it (SOLID principles promote reusability).

Example: `Logger` Class

```
public class Logger  {
   public void Log(string message)    {
      Console.WriteLine(message);
   }

   public void Info(string message)    {
      Console.WriteLine($"Info: {message}");
   }

   public void Debug(string message)   {
      Console.WriteLine($"Debug: {message}");
   }
}
```

Now, some developers want to change debug messages to suit their needs. For example, they want to start debugging messages with "`Dev Debug ->`". So, to satisfy their need, you need to edit the code of the `Logger` class and either create a new method for them or modify the existing `Debug()` method. If you change the existing `Debug()` method then the other developers who don't want this change will also be affected.

One way to use OCP and solve this problem is to use class based-inheritance (**polymorphism) and override methods**. You can **mark all the methods of the `Logger` class as `virtual`** so that if somebody wants to **change any of the methods then they can inherit the `Logger` class into a new class and override it**.

Example: Virtual Methods

```
public class Logger {
    public virtual void Log(string message) {
        Console.WriteLine(message);
    }

    public virtual void Info(string message){
        Console.WriteLine($"Info: {message}");
    }

    public virtual void Debug(string message){
        Console.WriteLine($"Debug: {message}");
    }
}
```

Now, a new class can inherit the `Logger` class and change one or more method behavior. The developers who wanted to change the debug message will create a new class, inherit the `Logger` class and override the `Debug()` method to display the message they wanted, as shown below.

Example: Modify Class by Overriding Methods

```
public class NewLogger : Logger {
   public override void Debug(string message)    {
      Console.WriteLine($"Dev Debug -> {message}");
   }
}
```

They will now use the above class to display debug messages they want without editing the source code of the original class.

Example:

```
public class Program  {
   public static void Main(string[] args)    {
      Logger logger = new Logger();
      logger.Debug("Testing debug");

      Logger newlogger = new NewLogger();
      newlogger.Debug("Testing debug ");
   }
}
```

Output:

```
Debug: Testing debug
Dev Debug -> Testing debug
```

Thus, OCP using inheritance makes it "**Open for extension and closed for modification**".

Let's take another example. The following is the `Course` class that we created in the previous SRP section.

Example:

```
public class Course  {
   public int CourseId { get; set; }
   public string Title { get; set; }
   public string Type { get; set; }
```

```
    public void Subscribe(Student std)    {
        Logger.Log("Starting Subscribe()");

        //apply business rules based on the course type live, online, offline, if any
        if (this.Type == "online")       {
            //subscribe to online course
        }
        else if (this.Type == "offline")   {
            //subscribe to offline course
        }

        // payment processing
        PaymentManager.ProcessPayment();

        //create CourseRepository class to save student and course into StudentCourse table

        // send confirmation email
        EmailManager.SendEmail();

        Logger.Log("End Subscribe()");
    }
}
```

We will have to edit the above `Course` class whenever there is a requirement of adding a new type of course. We will have to add one more if condition or switch cases to process the course type. Also, the above `Course` class does not follow the Single Responsibility Principle because if there is any change in the process of subscribing to courses or need to add new types of courses, then we will have to change the `Course` class.

7.2.1 Applying OCP

To apply OCP to our `Course` class, abstract class-based inheritance is more suitable. We can create an abstract class as a base class and then create a new class for each type of course and implement the `Subscribe()` method in each class which will do all the necessary subscription steps, as shown below.

Example: A Class follows OCP

```
public abstract class Course {
    public int CourseId { get; set; }
    public string Title { get; set; }

    public abstract void Subscribe(Student std);
}

public class OnlineCourse : Course {
    public override void Subscribe(Student std)    {
        //write code to subscribe to an online course
```

```
    }
}

public class OfflineCourse : Course {
    public override void Subscribe(Student std)    {
        //write code to subscribe to a offline course
    }
}
```

As you can see, the `Course` class is now an abstract class where the `Subscribe()` method is an abstract method that needs to be implemented in a class that inherits the `Course` class. This way there is a separate `Subscribe()` function for separate course types (Separation of concerns). You can create a new class for a new type of course in the future that inherits the `Course` class. That way, you don't have to edit the existing classes.

You can now subscribe a student to a course, as shown below:

```
public class Program {
    public static void Main(string[] args)    {
        Student std = new Student() { FirstName = "Steve", LastName = "Jobs" };
        Course onlineSoftwareEngCourse = new OnlineCourse() { Title = "Software Engneering" };
        onlineSoftwareEngCourse.Subscribe(std);
    }
}
```

7.2.2 Advantages of OCP

1. Minimize the possibilities of error by not modifying existing classes.
2. Easily add new functionalities by adding new classes wherein no current functionality depends on the new classes.
3. Promote the Single Responsibility Principle
4. Unit test each class

7.3 Liskov Substitution Principle (LSP)

Liskov Substitution Principle was introduced by Barbara Liskov in 1987. She described this principle in mathematical terms as below:

Let $\varphi(x)$ be a property provable about objects x of type T. Then $\varphi(y)$ should also be true for objects y of type S where S is a subtype of T.

LSP guides how to use inheritance in object-oriented programming. It is about subtyping, and how to correctly derive a type from a base type. Robert Martin explains LSP as below:

"Subtypes must be substitutable for their base type."
(i.e., A derived class must be correctly substitutable for its base class.)

Here, the type can be **interface**, **class**, or **abstract** class.

Let's simplify it further. **A derived class must be correctly substitutable for its base class.** When you derived a class from a base class then the **derived class should correctly implement all the methods of the base class**. It should not remove some methods by throwing `NotImplementedException`.

Consider the following `IMyCollection` **interface** which can be implemented to create any type of collection class.

```
//Based Class : Interface Class
public  interface  IMyCollection {
   void Add(int item);
   void Remove(int item);
   int Get(int idex);
}

//Derived Class : inheritance Class
public class MyReadOnlyCollection  implements  IMyCollection  {
   private IList _collection;

   public MyReadOnlyCollection(IList<int> col)    {
      _collection = col;
   }
   public void Add(int item)    {
      throw new NotImplementedException();
   }

   public int Get(int index)    {
      return _collection[index];
   }

   public void Remove(int item)    {
      throw new NotImplementedException();
   }
}
```

The above example **violates** the **Liskov Substitution principle** because the `MyReadOnlyCollection` class implements the `IMyCollection` interface but it throws `NotImplementedException` for two methods `Add()` and `Remove()` because the `MyReadOnlyCollection` class is for the read-only collection so you cannot add or remove any item. **LSP suggests that the subtype must be substitutable for the base class or base interface**. In the above example, **we should create another interface for read-only collection without `Add()` and `Remove()` methods.**

Let's understand what is the meaning of "A derived class should correctly implement methods of a base class"?

Consider the following `Rectangle` class:

```
//Based Class 1: Rectangle
public class Rectangle {
   public  int  Height;
   public  int  Width;
   public abstract getHeight(){ };
   public abstract setWidth(){ };
}
```

Mathematically, a square is the same as a rectangle that has four equal sides. We can use **inheritance "is-a" relationship** here. **A square is a rectangle**. The `Square` class can inherit the `Rectangle` class with equal height and width, as shown below.

```
//Derived Class 2: Square
class Square extends Rectangle {
   private int _height;
   private int _width;

   @ override
   public int getHeight ()  {
       return  this._height;
   }
   public int   setHeight(int value) {
        this._height = value;
         this._width = value;
      }
   }
   @ override
   public  int  getWidth()   {
```

```
    return    this._width;
  }
    public void  setWidth (int value)       {
       this._width = value;
       this._height = value;
    }
  }
}
```

The following calculates the area of a rectangle:

```
// Class 3: AreaCalculatior
public class AreaCalculator {
  public static int CalculateArea(Rectangle  r)    {
    return r.getHeight() * r.getWidth();
  }
}
```

Now, the following returns the wrong results:

Example: LSP Violation

```
// Class 4: LSP Test class
public class LSPTest
{
  public static void main(String[] arg)    {
       Rectangle  sqr1 = new Square();
       sqr1.setHeight ( 6 );
       sqr1.setWidth ( 8 );

       System.out.println ("Area of  Square 1  :"  +  AreaCalculator.CalculateArea(sqr1));
       //returns 64

       Rectangle sqr2 = new Square();
       sqr2.setHeight ( 8 );
       sqr2.setWidth ( 6 );

       System.out.println ("Area of  Square 2  :"  +AreaCalculator.CalculateArea(sqr2));
       //returns 36    }
}
```

LSP says that the derived class should correctly implement the base class methods. Here, **the square class is not a subtype of the rectangle class because it has equal sides.** So,

only one property is needed instead of two properties, height, and width. It creates confusion for the users of the class and might give the wrong result.

7.4 Interface Segregation Principle (ISP)

Interface Segregation Principle (ISP) is the fourth principle of SOLID principles. It can be used in conjunction with LSP.

Interface Segregation Principle (ISP) states that:

"Clients should not be forced to depend on methods they do not use."

Now, who is the client and what and whose methods it is talking about?

Here, **a client is a code that calls the methods of a class** with an instance of the interface. For example, a class implements an interface that contains 10 methods. Now, you create an object of that class with a variable of that interface and call only 5 methods for the functionality you wanted and never call the other 5 methods. So, this means that **the interface contains more methods that are not used by all client codes.** It is **called a fat interface**. **ISP suggests** segregating that **interface into two or more interfaces** so that **a class can implement the specific interface that it requires.**

Let's use the following interface to learn ISP in detail:

Example: Fat Interface

```
// Class 1: Interface class  - IStudentRepository
Public  interface  IStudentRepository
{
   void AddStudent(Student std);
   void EditStudent(Student std);
   void DeleteStudent(Student std);

   void AddCourse(Course cs);
   void EditCourse(Course cs);
   void DeleteCourse(Course cs);

   bool SubscribeCourse(Course cs);
   bool UnSubscribeCourse(Course cs);
   IList<Student>  GetAllStudents();
   IList<Student> GetAllStudents(Course cs);
   IList<Course> GetAllCourse();
   IList<Course> GetAllCourses(Student std);
}
```

// Class 2: Derived class

```
public class StudentRepository  implements IStudentRepository
{
   public void AddCourse(Course cs)    {
      //implementation code removed for better clarity
   }

   public void AddStudent(Student std)    {
      //implementation code removed for better clarity
   }

   public void DeleteCourse(Course cs)    {
      //implementation code removed for better clarity
   }

   public void DeleteStudent(Student std)    {
      //implementation code removed for better clarity
   }

   public void EditCourse(Course cs)    {
      //implementation code removed for better clarity
   }

   public void EditStudent(Student std)    {
      //implementation code removed for better clarity
   }

   public IList<Course> GetAllCourse()    {
      //implementation code removed for better clarity
   }

   public IList<Course> GetAllCourses(Student std)    {
      //implementation code removed for better clarity
   }
   public IList<Student> GetAllStudents()    {
      //implementation code removed for better clarity
   }

   public IList<Student> GetAllStudents(Course cs)    {
      //implementation code removed for better clarity
   }

   public bool SubscribeCourse(Course cs)    {
      //implementation code removed for better clarity
   }

   public bool UnSubscribeCourse(Course cs)    {
      //implementation code removed for better clarity
   }
}
```

The above `IStudentRepository` interface contains 12 methods for different purposes. The `StudentRepository` class implements the `IStudentRepository` interface.

Now, after some time, you observe that not all instances of the `StudentRepository` class call all the methods. Sometimes it calls methods that perform student-related tasks or sometimes calls course-related methods. Also, the `StudentRepository` does not follow the single responsibility principle because you may need to edit its code if student related as well as course-related business logic changes.

7.4.1 Applying ISP

To apply ISP to the above problem, we can **split our large interface** `IStudentRepository` and create another interface `ICourseRepository` with all course-related methods, as shown below.

Example: Interfaces after apllying ISP

```
// Class 1: Interface class  - IStudentRepository
public interface IStudentRepository
{
   void AddStudent(Student std);
   void EditStudent(Student std);
   void DeleteStudent(Student std);

   bool SubscribeCourse(Course cs);
   bool UnSubscribeCourse(Course cs);
   IList<Student> GetAllStudents();
   IList<Student> GetAllStudents(Course cs);
}

// Class 2: Interface class  - ICourseRepository
public interface ICourseRepository
{
   void AddCourse(Course cs);
   void EditCourse(Course cs);
   void DeleteCourse(Course cs);

   IList<Course> GetAllCourse();
   IList<Course> GetAllCourses(Student std);
}
```

Now, we can create two concrete classes that implement the above two interfaces. **This will automatically support SRP and increase cohesion.**

ISP is not specific to interfaces only but **it can be used with abstract classes** or any class that provides some services to the client code.

ISP helps in implementing **Liskov Substitution Principle**, **increasing cohesion** that in turn **supports the Single Responsibility Principle**.

The following **code smells detects ISP violation**:

1. When you have large interfaces.
2. When you implement an interface in a concrete class where some methods do not have any implementation code or throw NotImplementedException.
3. When you call only a small set of methods of a larger interface.

The **solution** to ISP violations:

1. Split large interfaces into smaller ones.
2. Inherit multiple small interfaces if required.
3. Use the adapter design pattern for the third-party large interface so that your code can work with the adapter.

7.5 Dependency Inversion Principle (DIP)

Dependency Inversion Principle is the last principle of SOLID principles. It helps in **loose coupling**. Dependency Inversion Principle states that:

"High-level modules should not depend on low-level modules. Both should depend on abstraction."

Now, the question is **what are high-level and low-level modules** and **what is an abstraction?**

A high-level module is a module (class) that uses other modules (classes) to perform a task. A low-level module contains a detailed implementation of some specific task that can be used by other modules. The **high-level modules** are generally the **core business logic of an application** whereas the **low-level modules** are **input/output, database, file system, web API**, or other external modules that interact with users, hardware, or other systems.

Abstraction is something that is not concrete. Abstraction **should not depend on detail but details should depend on abstraction**. For example, an abstract class or

interface contains methods declarations that need to be implemented in concrete classes. Those concrete classes depend on the abstract class or interface but not vice-versa.

Now, **how do we know a class depends on another class?**

You can identify a class is depends on another class **if it creates an object of another class.** You may require adding the reference of the namespace to compile or run the code. Let's use the following example to understand the DIP:

Example: Classes without Dependency Inversion Principle (DIP)

```
//class 1 : Student Class
public class Student
{
   public int StudentId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public DateTime DoB { get; set; }

   //tight coupling
   private  StudentRepository  _stdRepo = new  StudentRepository();

   public Student()  {

   }

   public void Save()   {
      _stdRepo.AddStudent(this);
   }
}

//Class 2: StudentRepository Class
public class StudentRepository  {
   public void AddStudent(Student std)    {
      //EF code removed for clarity
   }

   public void DeleteStudent(Student std)   {
      //EF code removed for clarity
   }

   public void EditStudent(Student std)    {
      //EF code removed for clarity
   }

   public IList<Student> GetAllStudents()    {
      //EF code removed for clarity
   }
}
```

The above `Student` class creates an object of the `StudentRepository` class for CRUD operation to a database. Thus, the `Student` class depends on the `StudentRepository` class for CRUD operations. **The `Student` class is the high-level module and the `StudentRepository` class is the low-level module.**

Here, the **problem is** that the `Student` class **creates an object of concrete** `StudentRepository` class using the new keyword and **makes both tightly coupled**. **This leads to the following problems:**

- **Creating objects using the new keyword** at **all places is repeated code**. The object creation is not in one place.
    - **Violation of** the ***Do Not Repeat Yourself*** **(DRY) principle**. If there is some change in the constructor of the StudentRepository class then we need to make the changes in all the places. If object creation is in one place then it would be easy to maintain the code.
- Creating an object using new also **make unit testing impossible**. We cannot unit test the Student class separately.
- The StudentRepository class is a **concrete class**, so **any changes in the class will require changing the Student class too**.

7.5.1 Applying Dependency Inversion Principle (DIP)

DIP says that high-level modules should not depend on the low-level module. Both should depend on abstraction. Here, **abstraction means use of interface or abstract class**. The following is the result of applying the DIP principle to the above example.

Example: Classes after applying DIP

```
//class 1 : Student Class
public class  Student
{
   public int StudentId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public DateTime DoB { get; set; }

   private IStudentRepository _stdRepo;

   public Student(IStudentRepository stdRepo)  {
      _stdRepo = stdRepo;
   }

   public void Save()   {
      _stdRepo.AddStudent(this);
```

```
    }
}

//Interface Class 2: StudentRepository Class
public  interface  IStudentRepository
{
    void AddStudent(Student std);
    void EditStudent(Student std);
    void DeleteStudent(Student std);

    IList<Student> GetAllStudents();
}

//Derived Class 3: StudentRepository Class
public class StudentRepository  implements  IStudentRepository
{
    public void AddStudent(Student std)    {
        //code removed for clarity
    }

    public void DeleteStudent(Student std)    {
        //code removed for clarity
    }

    public void EditStudent(Student std)    {
        //code removed for clarity
    }

    public IList<Student> GetAllStudents()    {
        //code removed for clarity
    }
}
```

The `StudentRepository` class above implements the `IStudentRepository` interface. Here, `IStudentRepository` is **an abstraction of CRUD operations for student-related data**. The `StudentRepository` class provides the implementation of those methods, so **it depends on the methods of** the `IStudentRepository` interface.

The `Student` class **does not create an object** of the `StudentRepository` class using the `new` keyword. The constructor requires a parameter of the `IStudentRepository` class which will be passed from the calling code. Thus, it also depends on the abstraction (interface) rather than the low-level concrete class (`StudentRepository`).

This will create loose coupling and also make each class unit testable. The caller of the `Student` class can pass an object of any class that implements the `IStudentRepository` interface and by so **not tied to the specific concrete class**.

```
//Test Class
public  class  Program
{
   public static void Main(string[] args)    {
                //for production
                Student  std1 = new Student(new StudentRepository);

                //for unit test
                Student std2 = new Student(new TestStudentRepository);
   }
}
```

Instead of creating manually, you **can use the factory class to create it**, so that **all the object creation will be in one place**.

Example: DIP using Factory Class

```
//using design pattern – Factory Class
public class RepositoryFactory
{
   public static IStudentRepository  GetStudentRepository()     {
      return new StudentRepository();
   }
   public static IStudentRepository  GetTestStudentRepository()    {
      return new TestStudentRepository();
   }
}

//Test Class
public class Program
{
   public static void Main(string[] args)    {
         //for production
          Student std1 = new Student(RepositoryFactory.GetStudentRepository());

         //for unit test
          Student std2 = new Student(RepositoryFactory.TestGetStudentRepository());
   }
}
```

It is recommended to use **Dependency Injection** and **IoC containers** for **creating and passing objects of low-level classes to high-level classes.**

Check Your Understanding on SOLID: Single Responsibility Principle

Q1. What are SOLD principles? Are SOLD patterns? What SOLD stands for?

Q2. What are the SOLD principles? Explain each principle guide for the OO programming.

Q3. What does the single responsibility principle (SRP) state for the object oriented programming? Explain with example.

Q4. How does the single responsibility principle (SRP) apply in the OO program?Explain with example.

Q5. What does the Separation of Concerns principle suggest for OO programming?

Q6. How does the SRP and the Separation of Concerns principle effect on cohesion and coupling of OO program?

Q7. What does the Open/Closed Principle (OCP) state for the object oriented programming? Explain with example.

Q8. How does the Open/Closed Principle (OCP) apply in the OO program? Explain with example.

Q9. What does the Liskov Substitution Principle guide for the object oriented programming? Explain with example.

Q10. What does Interface Segregation Principle (ISP) guide for the object oriented programming? Explain with example.

Q11. What does Dependency Inversion Principle guide for the object oriented programming? Explain with example.

SUMMARY

- A simple strategy for implementing Object-Oriented Concepts, Design and Implementation.
- Relationship between classes such as inheritance, polymorphism, composition, aggregation and association relationships are discussed with implementation strategies.
- Association classes should be implemented as classes. This involves introducing additional associations connecting the association class to the original classes.
- Strategy for implementing associations can be either unidirectional or bidirectional, depending on how the association needs to be navigated. Bidirectional implementations of associations need to maintain the referential integrity of the references implementing a link. A strategy for robustness is to assign the responsibility for maintaining references to only one of the classes involved.
- Qualified associations can be implemented by providing some form of data structure mapping qualifiers onto the objects they identify.
- Constraints in a model can be explicitly checked in code, but often it is only worth doing this for the preconditions of operations.
- The information provided in a statechart can be used to guide the implementation of a class. A simple strategy is to enumerate the possible states, store the current state and structure the implementation of the operations so that the effect of a message in each state is made clear, perhaps by using a switch statement.
- The Interaction Diagrams such as the sequence and collaboration diagram depict the relationships between the objects in a system and objects work together to perform a particular task. It defines a functionality of the system by demonstrating a set of chronological interactions between objects. It is basically used to demonstrate how objects interrelate to perform the behaviour of a specific use case.
- The implementation of the sequence and collaboration diagrams involves the classes and the methods that eventuality of a complex behavior of the elements such as actors, objects and their communication links, and messages.

********** End of lecture 1 **********

The more that you read, the more things you will know.

The more that you learn, the more places you will go.

Dr. Sesus