OOP Analysis & Design - Exemplified by UML

[Acknowledgement: This is a Compendium on Object-Oriented analysis & design by UML – materials adapted from a variety of web sources]

Unified Modelling Language (UML) is a standardized modelling language consisting of an integrated set of diagrams, a most widely used tool to analyse and design of an object oriented software systems by diagrammatic representation. UML facilitates software developers to specify, construct, visualise and document the artifacts of software systems as well as business modelling. According to the Object Management Group (OMG), UML meets requirements of scalability, robustness, security, extendibility, and other characteristics before implementation in code makes changes difficult to perform.

In this compendium, we will give you detailed ideas with examples about how UML could be used for OO analysis and design.

UML Behavioural Diagrams [1]

A. USE CASE Diagram

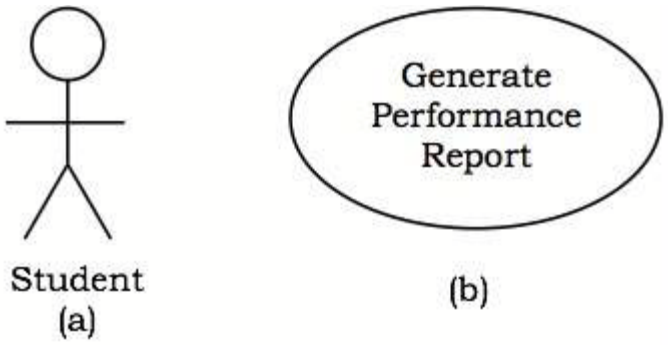
(a) Use case

A use case describes the sequence of actions a system performs yielding visible results. It shows the interaction of things outside the system with the system itself. Use cases may be applied to the whole system as well as a part of the system.

(b) Actor

An actor represents the roles that the users of the use cases play. An actor may be a person (e.g. student, customer), a device (e.g. workstation), or another system (e.g. bank, institution).

The following figure shows the notations of an actor named Student and a use case called Generate Performance Report.



(c) Use case diagrams

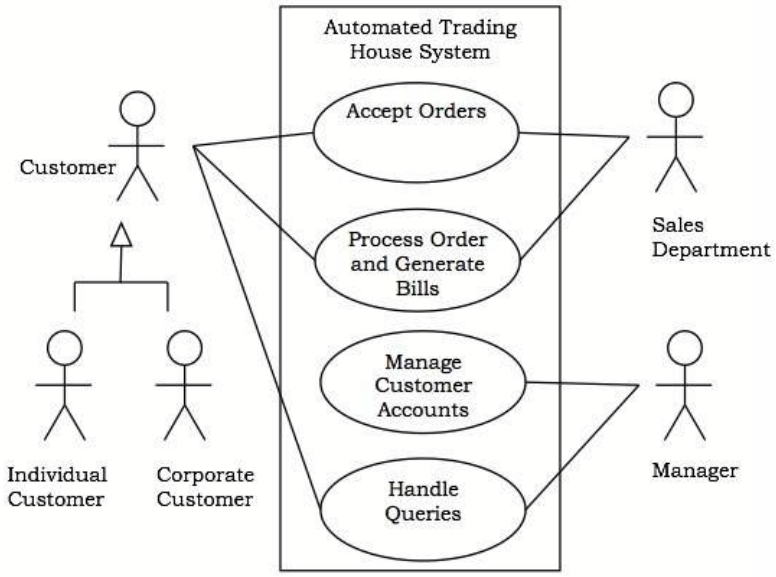
Use case diagrams present an outside view of the manner the elements in a system behave and how they can be used in the context.

Use case diagrams comprise of:

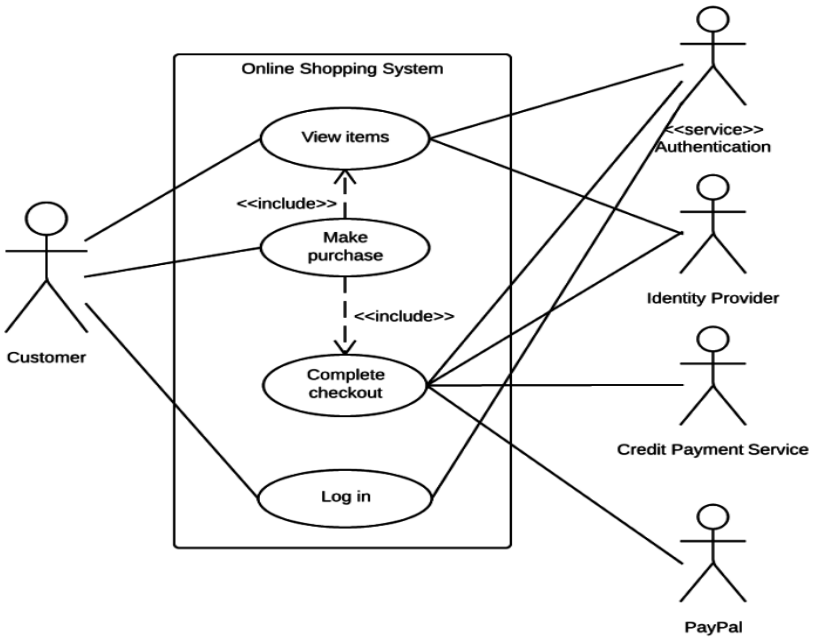
- Use cases
- Actors
- Relationships like dependency, generalization, and association

Let us consider an Automated Trading House System. We assume the following features of the system:

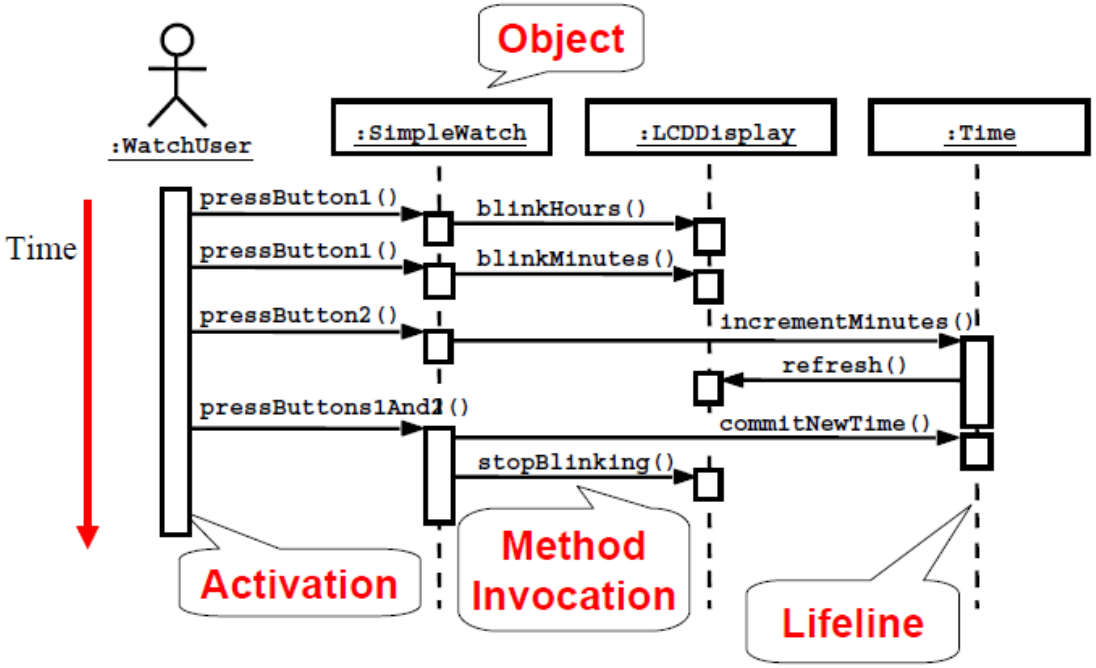
- The trading house has transactions with two types of customers, individual customers and corporate customers.
- Once the customer places an order, it is processed by the sales department and the customer is given the bill.
- The system allows the manager to manage customer accounts and answer any queries posted by the customer.



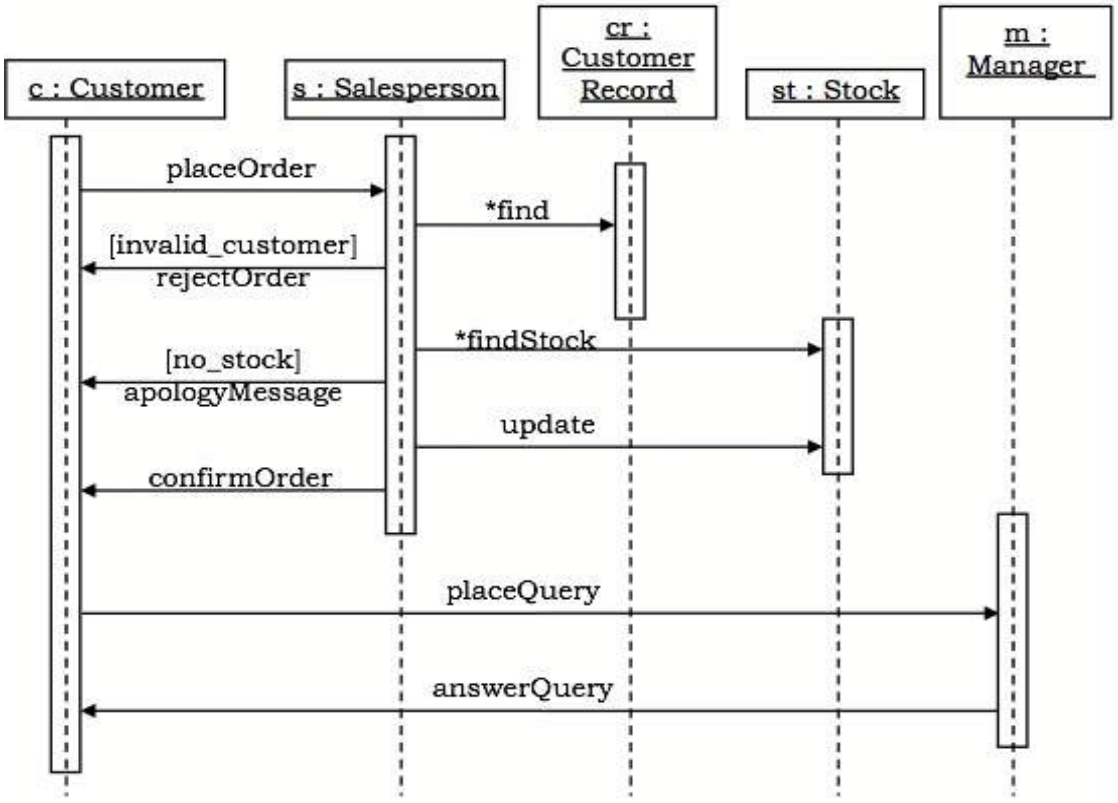
*** Use case diagram for Online Shopping System as shown in Figure below:**



B. INTERACTION DIAGRAM – SEQUENCE DIAGRAM



A sequence diagram for the Automated Trading House System is shown in the following figure.



C. ACTIVITY DIAGRAM

An activity diagram depicts the flow of activities which are ongoing non-atomic operations in a state machine. Activities result in actions which are atomic operations.

[**NOTE:** In concurrent programming, an **operation** (or set of **operations**) is **atomic**, linearizable, indivisible or uninterruptible if it appears to the rest of the system to occur instantaneously.]

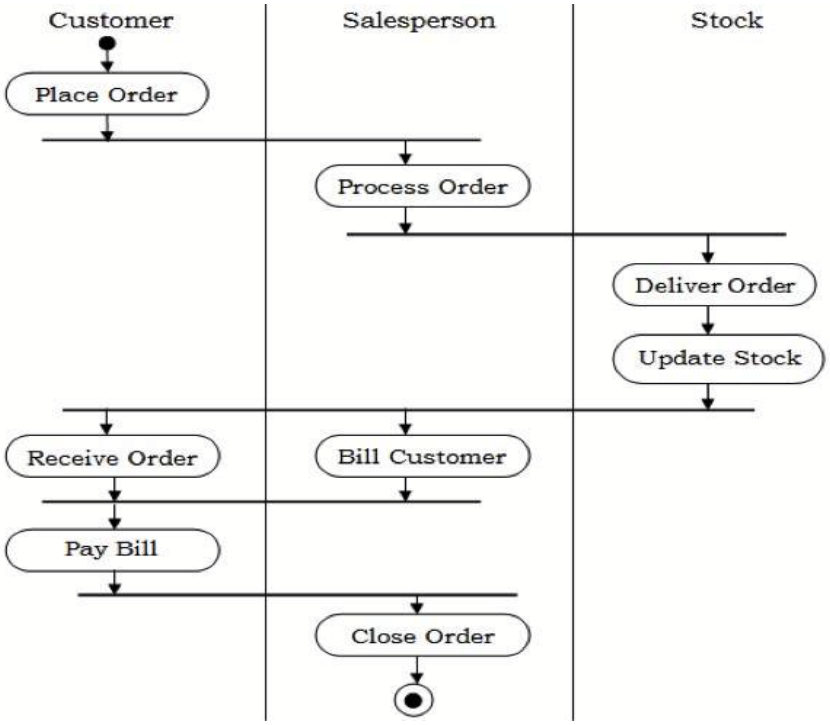
Activity diagrams comprise of:

- Activity states and action states
- Transitions
- Objects

Activity diagrams are used for modeling:

- workflows as viewed by actors, interacting with the system.
- details of operations or computations using flowcharts.

The following figure shows an activity diagram of a portion of the Automated Trading House System.



Some useful notations:

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

Class Structure

Each class can have a name, a list of attributes and a list of operations, as shown below. Each row represents a compartment.

Class Name	<<stereotype>> **Car**
Attributes	- numDoors : int - license: String
Operations	+ Car() + getNumDoors() : int + setNumDoors(doors: int) + getLicense() : String + setLicense(theLicense: String)

The stereotype is a word enclosed in « and » characters (called guillemets) or if those characters aren't available, simply << and >>. Example stereotypes that you have seen are <<interface>> and <<enumeration>>.

Multiplicities

Common multiplicities are:

1 (or 1..1) – There is a one to one relationship. For example, each Car has only one owner.

0..1 – There can be 0 or 1 item.

1..* - There can be a reference to one or more items.

*** (or 0..*)** - There can be a reference to many items or no item is referenced. For example, each Person can have zero or more cars.

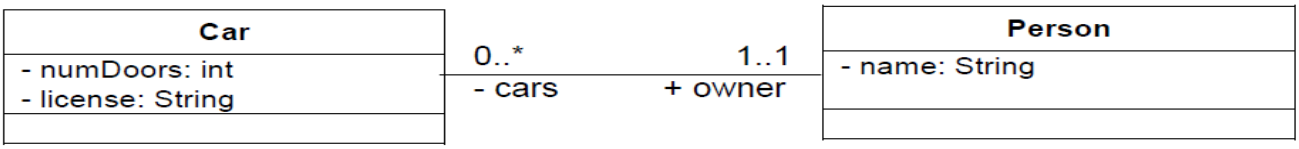
Notation

- ***** ⇒ **0, 1, or more**
- **5** ⇒ **5 exactly**
- **5..8** ⇒ **between 5 and 8, inclusive**
- **5..*** ⇒ **5 or more**

Associations

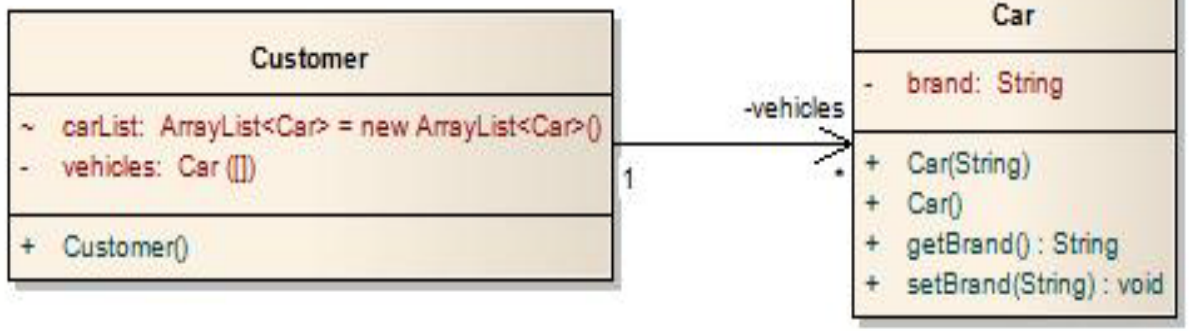

```
public class Customer {
private String name;
private String address;
private String contactNumber;
}
public class Car {
    private String modelNumber;
    private Customer owner;
}
```

This type of association uses a solid line to connect the two classes. Each end can include multiplicities, explained below, and the role that the class plays in the association.



If you were to write these out as Java code, you would have the following two definitions:

```
public class Car {
    private int numDoors;
    private String license;
    public Person owner;
}

public class Person {
    private String name;
    private Car[] cars;
}
```



Car.java

```
public class Car {
    private String brand;
    public Car(String brands){
        this.brand = brands;
    }
    public Car() {
    }
    public String getBrand() {
        return brand;
    }
    public void setBrand(String brand) {
        this.brand = brand;
    }
}
```

Customer.java

```
public class Customer {
    private Car[] vehicles;
    ArrayList<Car> carList = new ArrayList<Car>();

public Customer(){
        vehicles = new Car[2];
        vehicles[0] = new Car("Audi");
        vehicles[1] = new Car("Mercedes");
        carList.add(new Car("BMW"));
        carList.add(new Car("Chevy"));
    }
}
```

UML

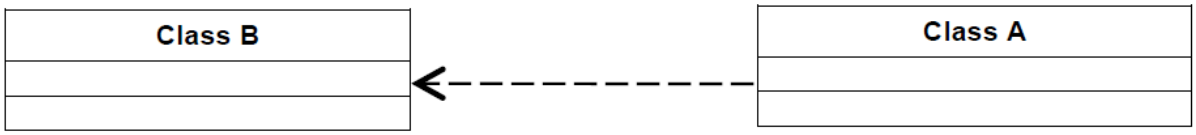
Java

```
class Pet {
   PetOwner myOwner;          // 1 owner for each pet
}
class PetOwner {
   Pet [ ] myPets;            // multiple pets for each owner
}
```

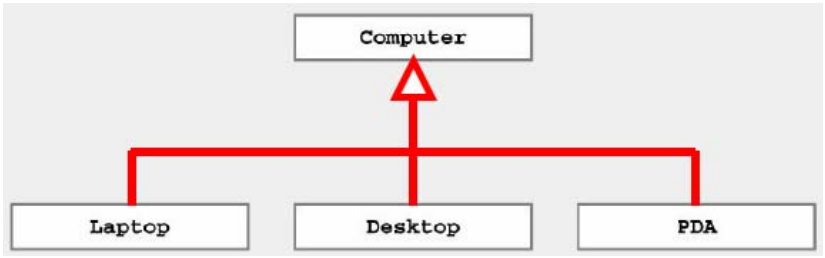
Dependency

A dependency states that somewhere in Class A, there is a reference to Class B. This differs from the main type of association in that the reference is either a local variable or an argument or return type to an operation.

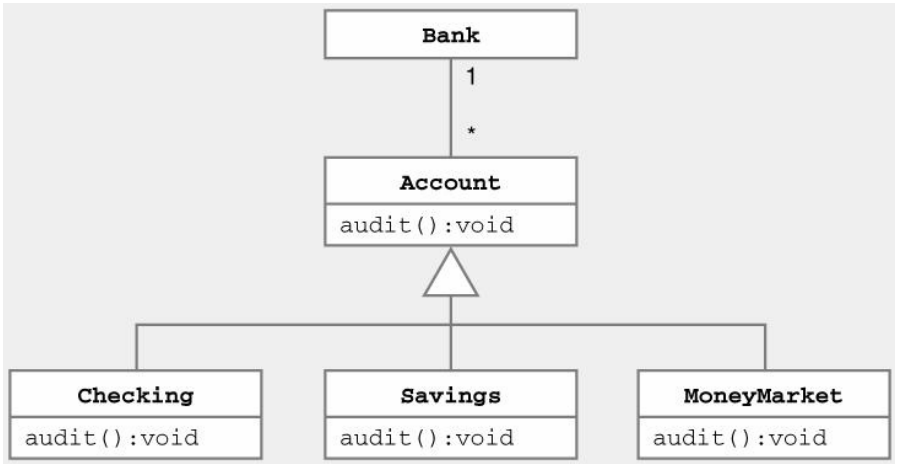
A dependency is drawn using a dashed line. The navigability symbol would be added to one end of the line as appropriate.



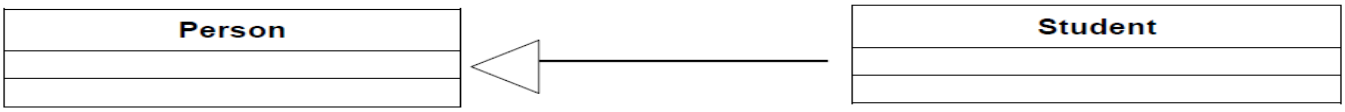
Inheritance / Generalisation



Laptop, Desktop, PDA inherit state & behavior from Computer



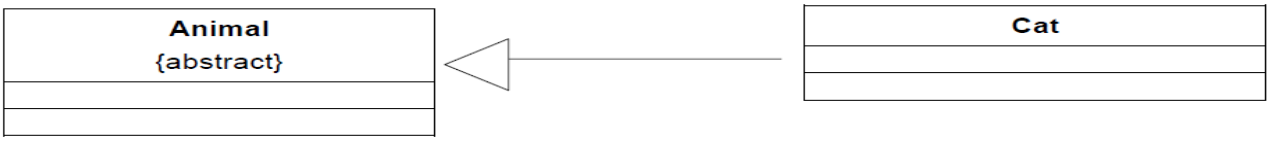
Inheritance represents the *is-a* relationship, for example, a Student is a special kind of Person.



The code for this diagram would be as follows:

```
public class Person {

}

public class Student extends Person {

}
```

The same type of diagram is used to show that one class extends an abstract class. For example, a Cat is a special kind of Animal, where an Animal is an abstract class.



The code for this diagram would be as follows:

```
public abstract class Animal {

}

public class Cat extends Animal {

}
```

Interfaces – Implementation

The Cat implements the Animal interface.

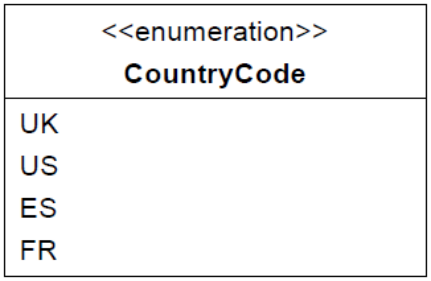
The corresponding code would be:

```
public interface Animal {
   public String sound();
}

public class Cat implements Animal {

   @Override
   public String sound(){
       return "miaow";
   }
}
```

Enumerations (enum)

In Java, an enumeration is a way to have a type that has a defined number of possible values.

In UML, we would write it as a class with two compartments. We would add the stereotype <<enumeration>> to the name; you might also see this as <<enum>>. The attributes compartment would list the possible values.

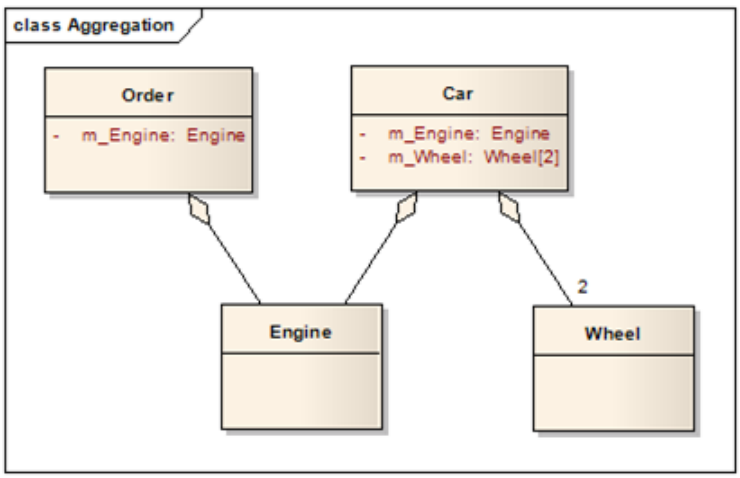


This would correspond to the Java code:

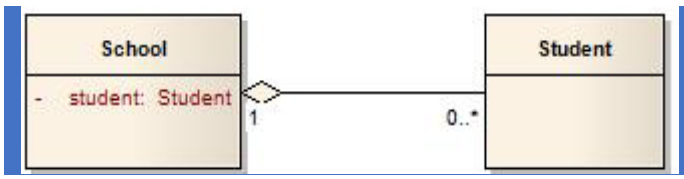
```
public enum CountryCode {
    UK,
    US,
    ES,
    FR
}
```

Aggregation (Shared Association)

In cases where there's a part-of relationship between ClassA (whole) and ClassB (part), we can be more specific and use the aggregation link instead of the association link, highlighting that ClassB can also be aggregated by other classes in the application (therefore aggregation is also known as shared association).



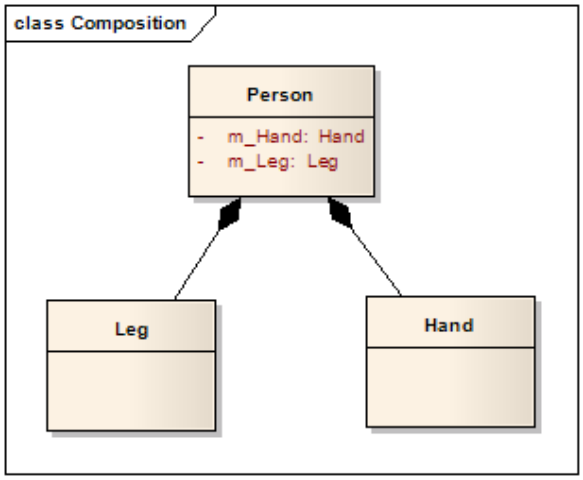
It's important to note that the aggregation link doesn't state in any way that ClassA owns ClassB nor that there's a parent-child relationship (when parent deleted all its child's are being deleted as a result) between the two. Actually, quite the opposite! The aggregation link usually used to stress the point that ClassA is not the exclusive container of ClassB, as in fact ClassB has another container.




```
public class Student {
}
public class School {
    private Student student;
}
```

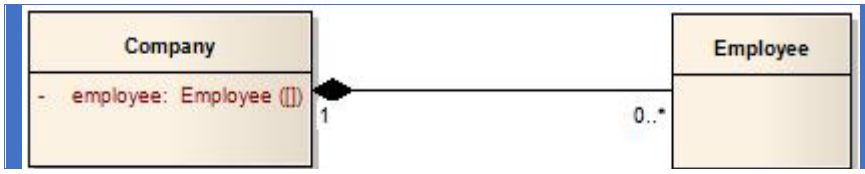
Composition (Not-Shared Association)

We should be more specific and use the composition link in cases where in addition to the part-of relationship between ClassA and ClassB - there's a strong lifecycle dependency between the two, meaning that when ClassA is deleted then ClassB is also deleted as a result



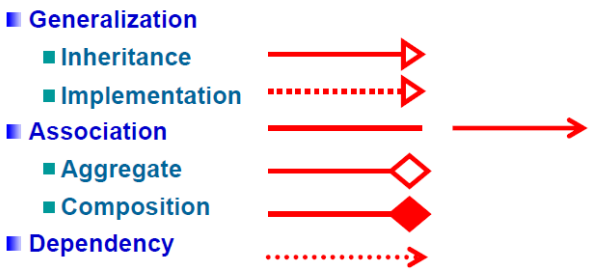
The composition link shows that a class (container, whole) has exclusive ownership over other class/s (parts), meaning that the container object and its parts constitute a parent-child/s relationship.

Aggregation is where an object owns objects, but may share these with other parts of the system. Composition is where an object owns a set of objects and the object is responsible for the lifetime of the objects.




```
public class Employee {
}
public class Company {
    private Employee[] employee;
}
```

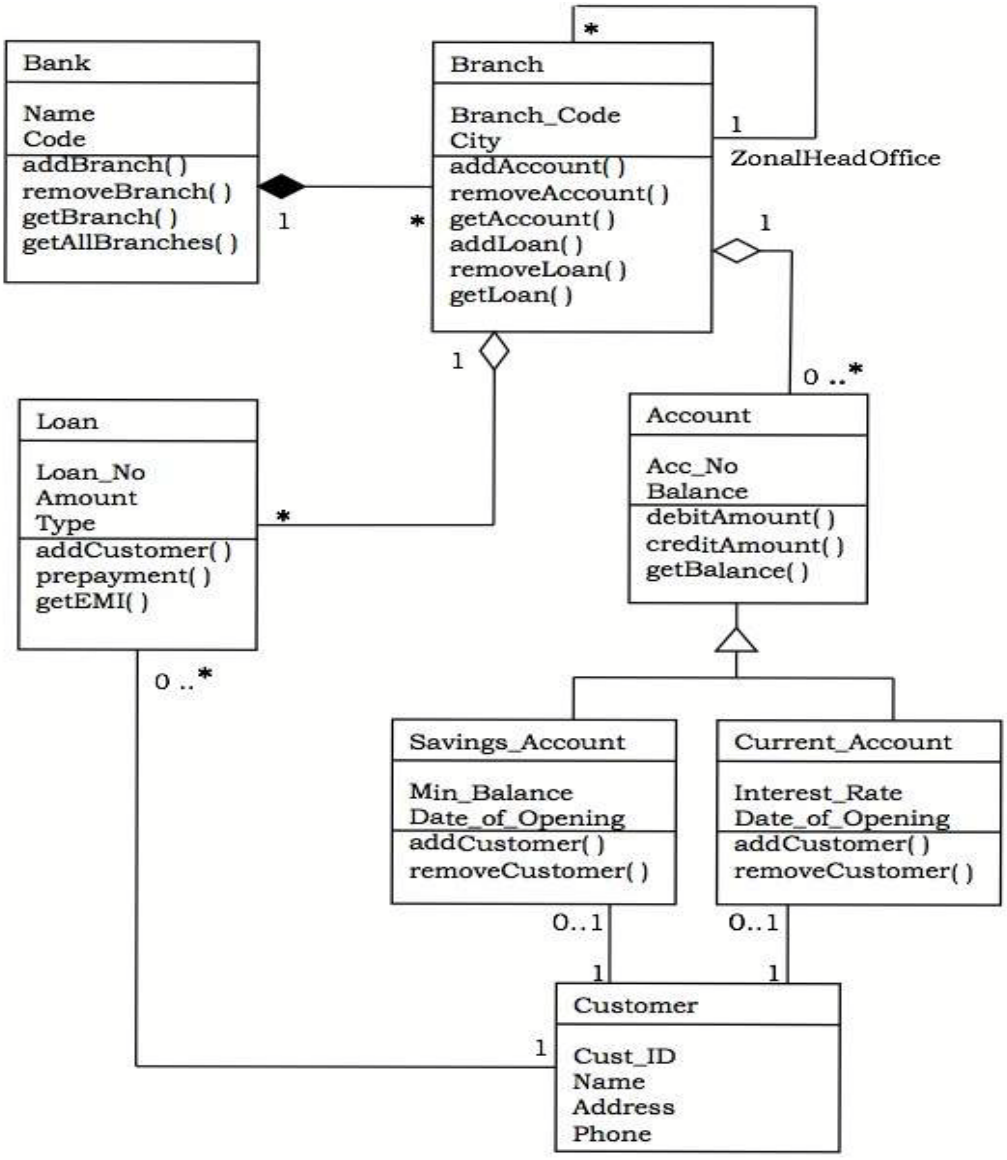
At a glance – Types of relationship



UML Structured Diagrams [1]

In the following diagram, **Classes given in the system are:**

Bank, Branch, Account, Savings Account, Current Account, Loan, and Customer.



Relationships:

- **A Bank "has–a" number of Branches** : composition, one–to–many
- **A Branch with role Zonal Head Office supervises other Branches** : unary association, one–to-many
- **A Branch "has–a" number of accounts** : aggregation, one–to–many

From the class Account, two classes have inherited, namely, Savings Account and Current Account.

- **A Customer can have one Current Account** : association, one–to–one
- **A Customer can have one Savings Account** : association, one–to–one
- **A Branch "has–a" number of Loans** : aggregation, one–to–many
- **A Customer can take many loans** : association, one–to–many

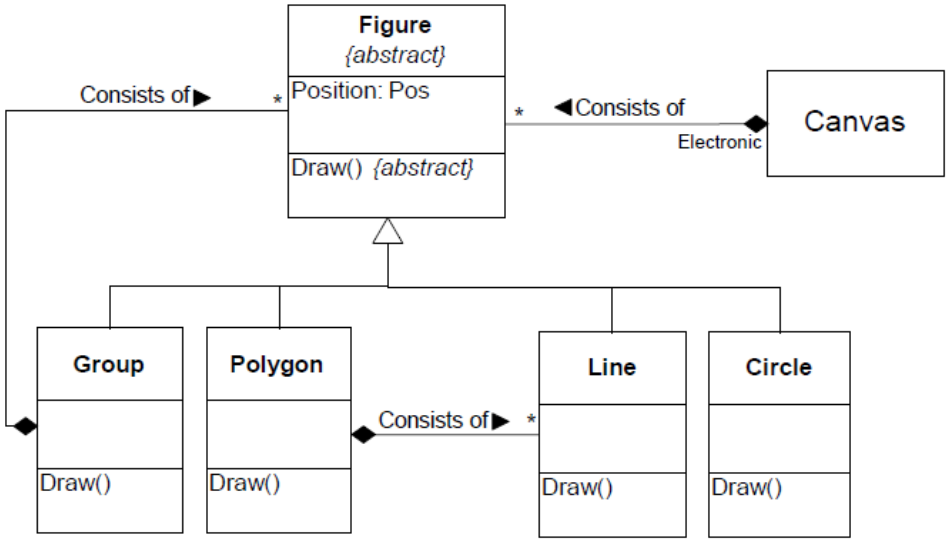
Example-1: Java Code for corresponding Class Diagram

```
class Clock { // name
  // state
  int seconds;
  int minutes;
  int hours;
  // behavior
  void start();
  void adjustTime();
  void reset();
}
```

Java Code

Class Diagram

Example-2: Aggregation & Generalisation

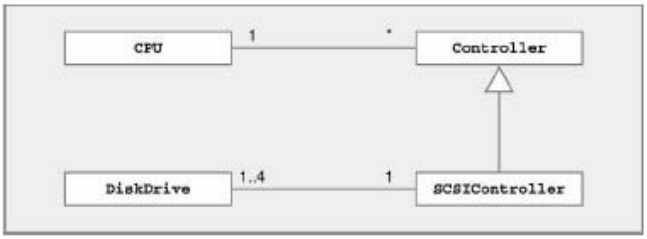



```
abstract public class Figure
{
  abstract public void Draw();
  Pos position;
}
public class Group extends Figure
{
  private FigureVector consist_of;
  public void Draw()
  {
    for (int i = 0; i < consist_of.size(), i++)
    {
      consist_of[i].draw();
    }
  }
}
public class Polygon extends Figure
{
  public void Draw()
  {
    /* something similar to group
       only using lines instead */
  }
}
```

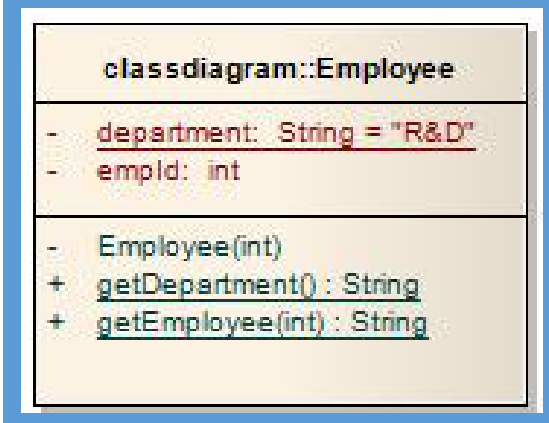
```
public class Line extends Figure
{
  public void Draw()
  {
    /* code to draw line */
  }
}
public class circle extends Figure
{
  public void Draw()
  {
    /* code to draw circle */
  }
}
```

Example-3: Association & Generalisation

```
class CPU {
   Controller [ ] myCtlrs;
}
class Controller {
   CPU myCPU;
}
class SCSIController extends Controller {
   DiskDrive [ ] myDrives = new DiskDrive[4];
}
Class DiskDrive {
    SCSIController mySCSI;
}
```

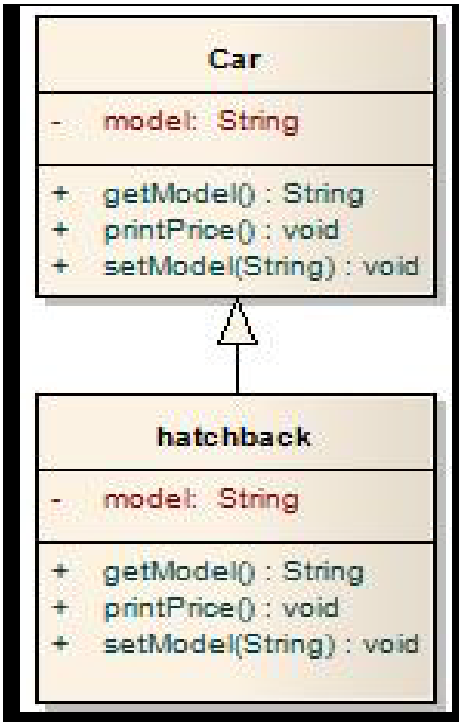


Example-4: Employee class



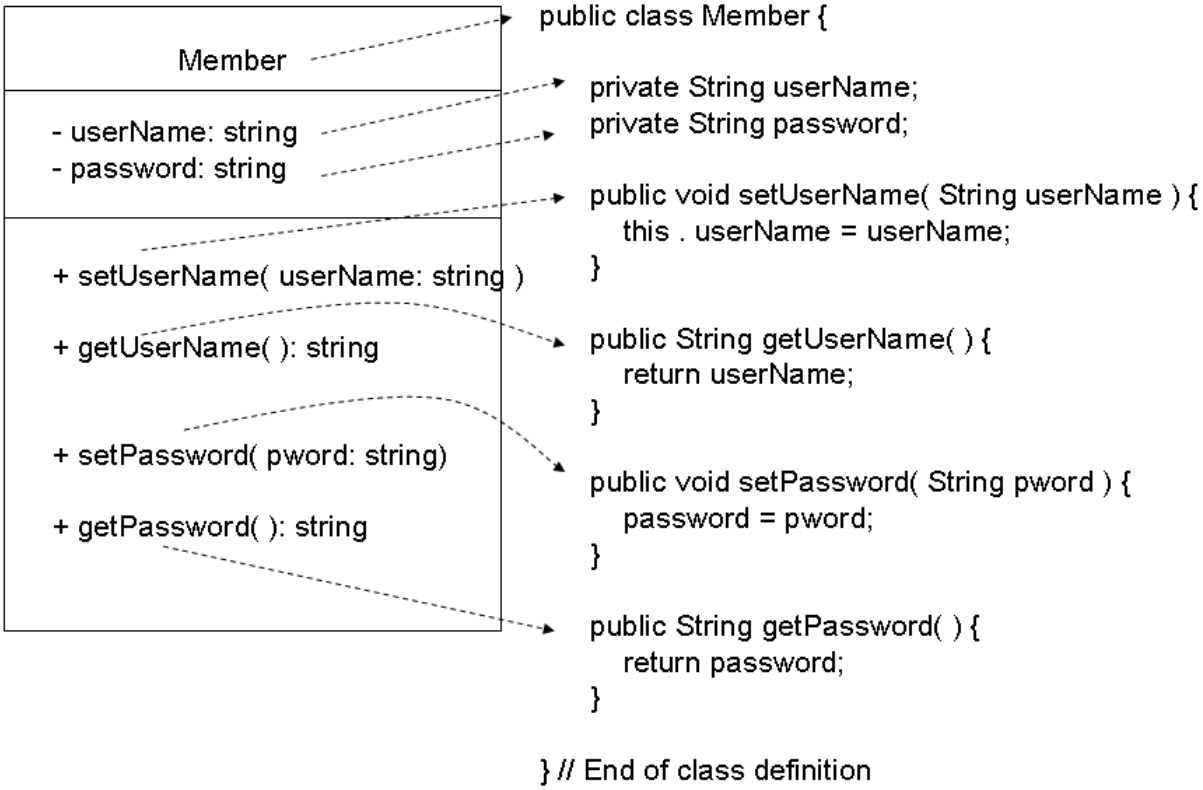

```
public class Employee {
    private static String department = "R&D";
    private int empId;
    private Employee(int employeeId) {
        this.empId = employeeId;
    }

public static String getEmployee(int emplId) {
        if (emplId == 1) {
            return "idiotechie";
        } else {
            return "Employee not found";
        }
    }

public static String getDepartment() {
        return department;
    }
}
```

Example-5: Generalisation




```
public class Car {
    private String model;
    public void printPrice() {
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
}
public class hatchback extends Car {
    private String model;
    public void printPrice() {
        System.out.println("Hatchback Price");
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
}
```

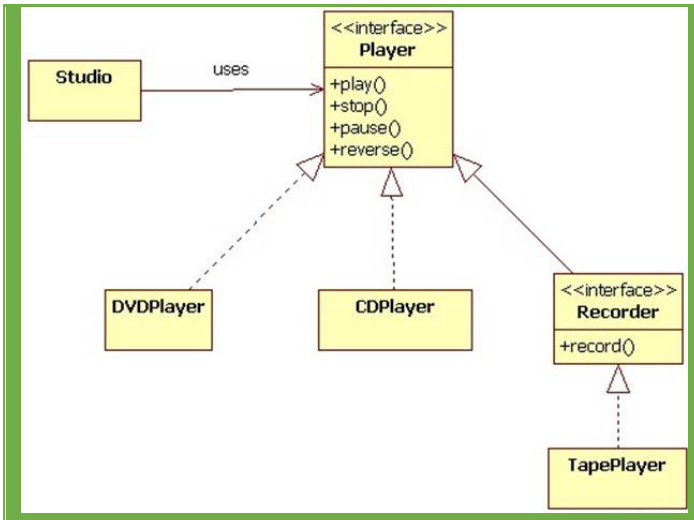
Example-6: Encapsulation



```
public class TestMember {

        public static void main( String[ ] args ) {

                Member member = new Member( ); // As before.
                member . setUserName( "Dylan" ); // As before.
                // Call the 'get' method of Member in a print statement.  The method
                // call returns a value.
                System.out.println( "The member's user name is: " +
                        member . getUserName( ) );

        } // End of definition of main.

} // End of class definition of TestMember.
```

Example-7: Interfaces and implementation

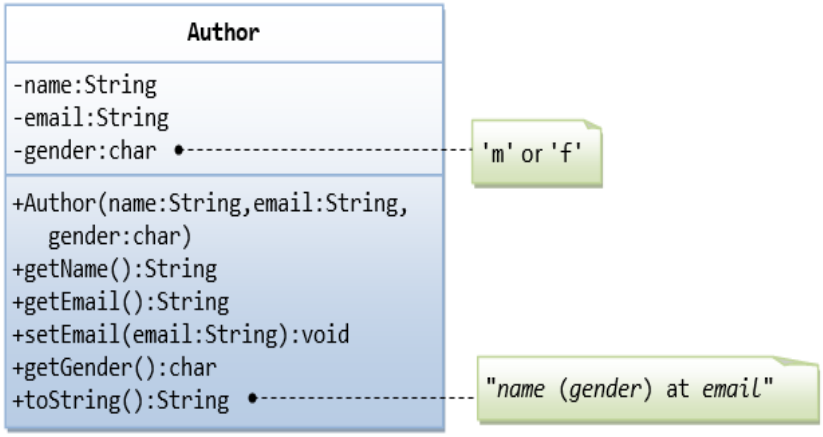
Here's the UML notation for two interfaces and three implementing classes [2]:




```
interface Player {
        void play();
        void stop();
        void pause();
        void reverse();
}
```

```
interface Recorder extends Player {
        void record();
}

class TapePlayer implements Recorder {
        public void play() { ... }
        public void stop() { ... }
        public void pause() { ... }
        public void reverse() { ... }
        public void record() { ... }
}

class DVDPlayer extends Player {
        public void play() {
                super.play(); // change state
                // DVD-specific play code goes here
        }
        // etc.
}
```

Example-8: The *Author* and *Book* Classes [3]



- 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`".

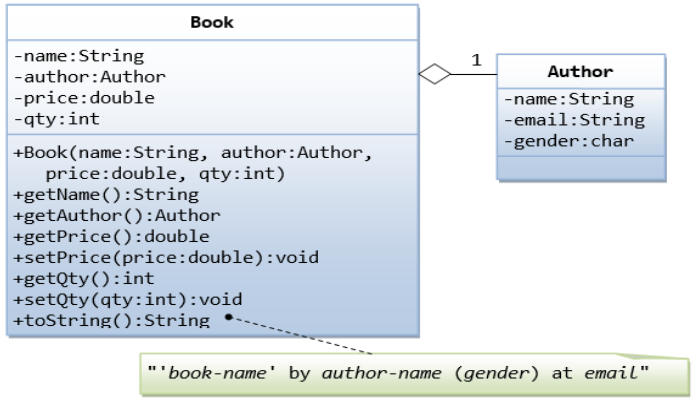
// Author.java

```
/*
 * 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'

   // The constructor
   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.
   public String getName() {
      return name;
   }
   public char getGender() {
      return gender;
   }
   public String getEmail() {
      return email;
   }
   public void setEmail(String email) {
      this.email = email;
   }

   // The toString() describes itself
   public String toString() {
      return name + " (" + gender + ")
at " + email;
   }
}
```

//TestAuthor.java

```
/*
 * A test driver for the Author
class.
 */

public class TestAuthor {

  public static void main(String[]
args) {
      // Test constructor and
toString()
      Author ahTeck = new Author("Tan
Ah Teck", "teck@nowhere.com", 'm');
      System.out.println(ahTeck);  //
toString()

      // Test Setters and Getters

ahTeck.setEmail("teck@somewhere.com")
;
      System.out.println(ahTeck);  //
toString()
      System.out.println("name is: "
+ ahTeck.getName());
      System.out.println("gender is:
" + ahTeck.getGender());
      System.out.println("email is: "
+ ahTeck.getEmail());
   }
}
```

A Book is written by one Author - Using an "Object" Member Variable



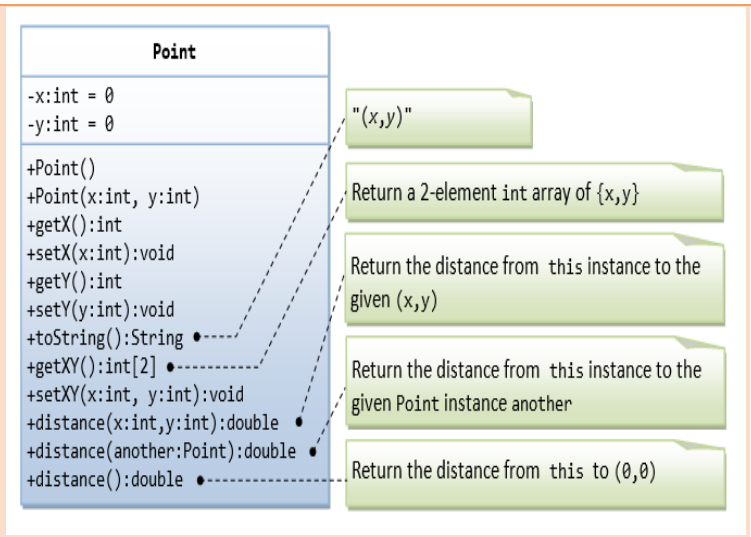
// Book.java

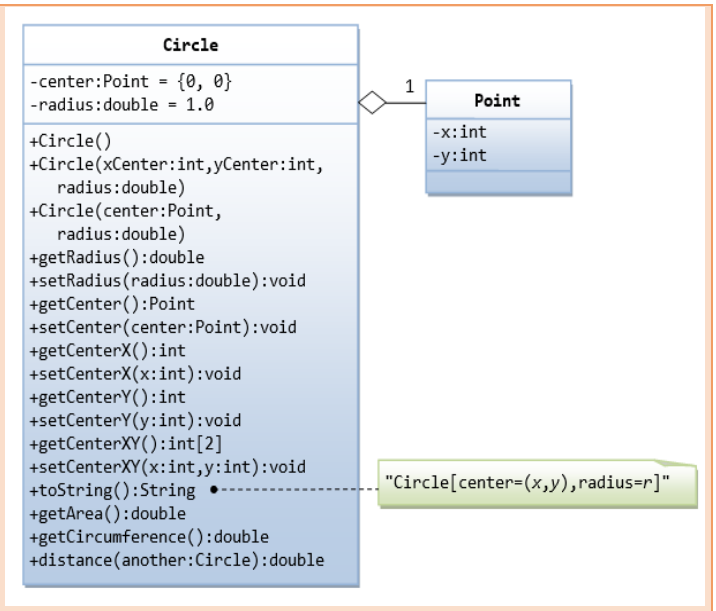
```
/*
 * The Book class models a book with one (and only one) author.
 */
public class Book {
   // The private instance variables
   private String name;
   private Author author;
   private double price;
   private int qty;

   // Constructor
   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
   public String getName() {
      return name;
   }
   public Author getAuthor() {
      return author;  // return member author, which is an instance of the class Author
   }
   public double getPrice() {
      return price;
   }
   public void setPrice(double price) {
      this.price = price;
   }
   public int getQty() {
      return qty;
   }
   public void setQty(int qty) {
      this.qty = qty;
   }

   // The toString() describes itself
   public String toString() {
      return "'" + name + "' by " + author;  // author.toString()
   }
}
```

// 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 ahTeck = new Author("Tan Ah Teck", "ahTeck@somewhere.com", 'm');
      System.out.println(ahTeck);  // Author's toString()

      // Test Book's constructor and toString()
      Book dummyBook = new Book("Java for dummies", ahTeck, 9.99, 99);
      System.out.println(dummyBook);  // Book's toString()

      // Test Setters and Getters
      dummyBook.setPrice(8.88);
      dummyBook.setQty(88);
      System.out.println(dummyBook);  // Book's toString()
      System.out.println("name is: " + dummyBook.getName());
      System.out.println("price is: " + dummyBook.getPrice());
      System.out.println("qty is: " + dummyBook.getQty());
      System.out.println("author is: " + dummyBook.getAuthor());  // invoke Author's toString()
      System.out.println("author's name is: " + dummyBook.getAuthor().getName());
      System.out.println("author's email is: " + dummyBook.getAuthor().getEmail());
      System.out.println("author's gender is: " + dummyBook.getAuthor().getGender());

      // 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'), // an anonymous Author's instance
            19.99, 8);
      System.out.println(moreDummyBook);  // Book's toString()
   }
}
```

Example-9:The *Point* and *Circle* Classes [3]





// Point.java

```
/*
 * The Point class models a 2D point at (x,
y).
 */
public class Point {
   // The private instance variables
   private int x, y;

   // The constructors (overloaded)
   public Point() {  // The default
constructor
      this.x = 0;
      this.y = 0;
   }
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }

   // The public getters and setters
   public int getX() {
      return this.x;
   }
   public void setX(int x) {
      this.x = x;
   }
   public int getY() {
      return this.y;
   }
   public void setY(int y) {
      this.y = y;
   }

   // Return "(x,y)"
   public String toString() {
      return "(" + this.x + "," + this.y +
")";
   }
```

// Circle.java

```
public class Circle {
   // The private member variables
   private Point center;  // Declare an
instance of the Point class
   private double radius;

   // Constructors
   public Circle() {
      this.center = new Point(); //
Construct a Point at (0,0)
      this.radius = 1.0;
   }
   public Circle(int xCenter, int yCenter,
double radius) {
      center = new Point(xCenter, yCenter);
// Construct a Point at (xCenter,yCenter)
      this.radius = radius;
   }
   public Circle(Point center, double
radius) {
      this.center = center;  // The caller
constructed an Point instance
      this.radius = radius;
   }

   // Getters and Setters
   public double getRadius() {
      return this.radius;
   }
   public void setRadius(double radius) {
      this.radius = radius;
   }
   public Point getCenter() {
      return this.center;  // return a
Point instance
```

```
   // Return a 2-element int array containing
x and y.
   public int[] getXY() {
      int[] results = new int[2];
      results[0] = this.x;
      results[1] = this.y;
      return results;
   }

   // Set both x and y.
   public void setXY(int x, int y) {
      this.x = x;
      this.y = y;
   }

   // Return the distance from this instance
to the given point at (x,y).
   public double distance(int x, int y) {
      int xDiff = this.x - x;
      int yDiff = this.y - y;
      return Math.sqrt(xDiff*xDiff +
yDiff*yDiff);
   }
   // Return the distance from this instance
to the given Point instance (called another).
   public double distance(Point another) {
      int xDiff = this.x - another.x;
      int yDiff = this.y - another.y;
      return Math.sqrt(xDiff*xDiff +
yDiff*yDiff);
   }
   // Return the distance from this instance
to (0,0).
   public double distance() {
      return Math.sqrt(this.x*this.x +
this.y*this.y);
   }
}
```

// TestPoint.java

```
/*
 * A Test Driver for the Point class.
 */
public class TestPoint {
   public static void main(String[] args) {
      // Test constructors and toString()
      Point p1 = new Point(1, 2);
      System.out.println(p1);  // toString()
      Point p2 = new Point();  // default
constructor
      System.out.println(p2);

      // Test Setters and Getters
      p1.setX(3);
      p1.setY(4);
      System.out.println(p1);  // run
toString() to inspect the modified instance
      System.out.println("X is: " +
p1.getX());
      System.out.println("Y is: " +
p1.getY());
   }
   public void setCenter(Point center) {
      this.center = center;
   }

   public int getCenterX() {
      return center.getX();  // Point's
getX()
   }
   public void setCenterX(int x) {
      center.setX(x);  // Point's setX()
   }
   public int getCenterY() {
      return center.getY();  // Point's
getY()
   }
   public void setCenterY(int y) {
      center.setY(y);  // Point's setY()
   }
   public int[] getCenterXY() {
      return center.getXY();  // Point's
getXY()
   }
   public void setCenterXY(int x, int y) {
      center.setXY(x, y);  // Point's
setXY()
   }

   public String toString() {
      return "Circle[center=" + center +
",radius=" + radius + "]";  // invoke
center.toString()
   }

   public double getArea() {
      return Math.PI * radius * radius;
   }

   public double getCircumference() {
      return 2.0 * Math.PI * radius;
   }

   // Return the distance from the center
of this instance to the center of
   // the given Circle instance called
another.
   public double distance(Circle another) {
      return
center.distance(another.center);  // Invoke
distance() of the Point class
   }
 }
```

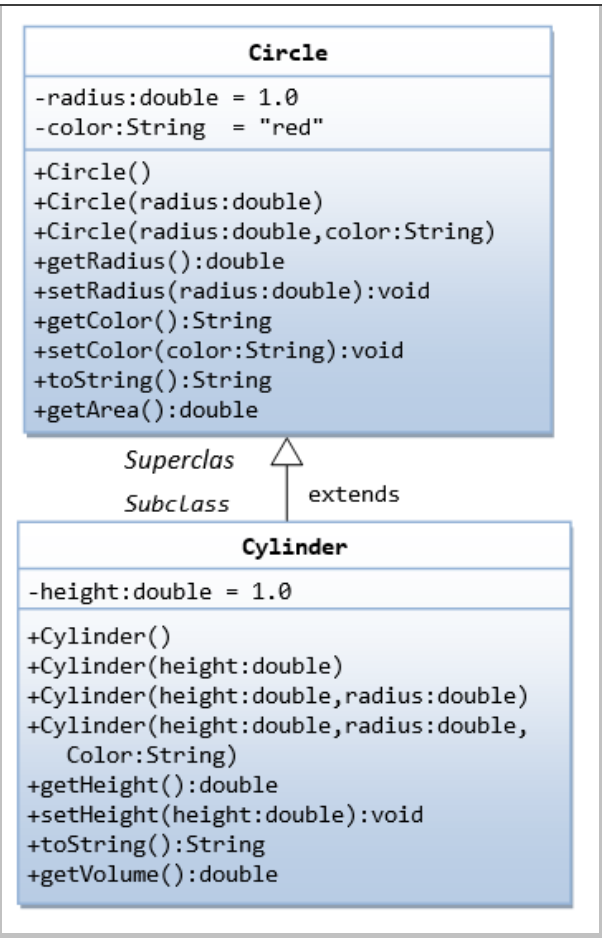
// TestCircle.java

```
public class TestCircle {
   public static void main(String[] args)
{
      // Test Constructors and toString()
      Circle c1 = new Circle();
```

```
      // Test setXY() and getXY()
      p1.setXY(5, 6);
      System.out.println(p1);  // toString()
      System.out.println("X is: " +
p1.getXY()[0]);
      System.out.println("Y is: " +
p1.getXY()[1]);

      // Test the 3 overloaded versions of
distance()
      p2.setXY(10, 11);
      System.out.printf("Distance is:
%.2f%n", p1.distance(10, 11));
      System.out.printf("Distance is:
%.2f%n", p1.distance(p2));
      System.out.printf("Distance is:
%.2f%n", p2.distance(p1));
      System.out.printf("Distance is:
%.2f%n", p1.distance());
   }
}
```

```
      System.out.println(c1);  // Circle's
toString()
      Circle c2 = new Circle(1, 2, 3.3);
      System.out.println(c2);  // Circle's
toString()
      Circle c3 = new Circle(new Point(4,
5), 6.6);   // an anonymous Point instance
      System.out.println(c3);  // Circle's
toString()

      // Test Setters and Getters
      c1.setCenter(new Point(11, 12));
      c1.setRadius(13.3);
      System.out.println(c1);  // Circle's
toString()
      System.out.println("center is: " +
c1.getCenter());  // Point's toString()
      System.out.println("radius is: " +
c1.getRadius());

      c1.setCenterX(21);
      c1.setCenterY(22);
      System.out.println(c1);  // Circle's
toString()
      System.out.println("center's x is: "
+ c1.getCenterX());
      System.out.println("center's y is: "
+ c1.getCenterY());
      c1.setCenterXY(31, 32);
      System.out.println(c1);  // Circle's
toString()
      System.out.println("center's x is: "
+ c1.getCenterXY()[0]);
      System.out.println("center's y is: "
+ c1.getCenterXY()[1]);

      // Test getArea() and
getCircumference()
      System.out.printf("area is: %.2f%n",
c1.getArea());
      System.out.printf("circumference is:
%.2f%n", c1.getCircumference());

      // Test distance()
      System.out.printf("distance is:
%.2f%n", c1.distance(c2));
      System.out.printf("distance is:
%.2f%n", c2.distance(c1));
   }
}
```

Example-10: Inheritance: The *Circle* and *Cylinder* Classes [3]



// Circle.java

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

   // Constructors
   public Circle() {
      this.radius = 1.0;
      this.color = "red";
   }
   public Circle(double radius) {
      this.radius = radius;
      this.color = "red";
   }
   public Circle(double radius, String
color) {
      this.radius = radius;
      this.color = color;
   }

   // Getters and Setters
   public double getRadius() {
      return this.radius;
   }
   public String getColor() {
```

// Cylinder.java

```
/*
 * A Cylinder is a Circle plus a height.
 */
public class Cylinder extends Circle {
   // private instance variable
   private double height;

   // Constructors
   public Cylinder() {
      super();  // invoke superclass'
constructor Circle()
      this.height = 1.0;
   }
   public Cylinder(double height) {
      super();  // invoke superclass'
constructor Circle()
      this.height = height;
   }
   public Cylinder(double height, double
radius) {
      super(radius);  // invoke superclass'
constructor Circle(radius)
      this.height = height;
   }
   public Cylinder(double height, double
radius, String color) {
      super(radius, color);  // invoke
superclass' constructor Circle(radius, color)
      this.height = height;
   }

   // Getter and Setter
   public double getHeight() {
      return this.height;
   }
   public void setHeight(double height) {
      this.height = height;
   }

   // Return the volume of this Cylinder
   public double getVolume() {
      return getArea()*height;   // Use
Circle's getArea()
   }

   // Describe itself
   public String toString() {
      return "This is a Cylinder";  // to be
refined later
   }
}
```

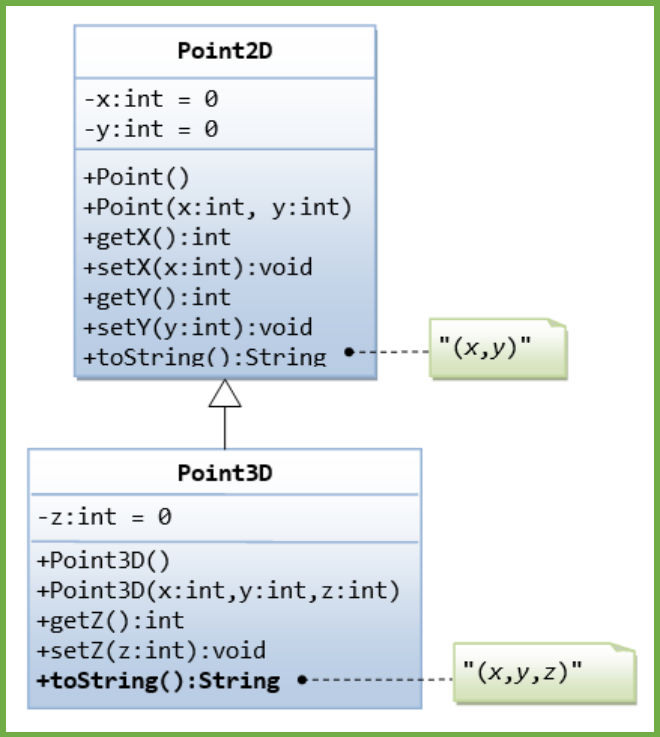
// TestCylinder.java

```
/*
```

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

   // Describle itself
   public String toString() {
      return "Circle[radius=" + radius +
",color=" + color + "]";
   }

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

```
 * A test driver for the Cylinder class.
 */
public class TestCylinder {
   public static void main(String[] args) {
      Cylinder cy1 = new Cylinder();
      System.out.println("Radius is " +
cy1.getRadius()
         + " Height is " + cy1.getHeight()
         + " Color is " + cy1.getColor()
         + " Base area is " + cy1.getArea()
         + " Volume is " + cy1.getVolume());

      Cylinder cy2 = new Cylinder(5.0, 2.0);
      System.out.println("Radius is " +
cy2.getRadius()
         + " Height is " + cy2.getHeight()
         + " Color is " + cy2.getColor()
         + " Base area is " + cy2.getArea()
         + " Volume is " + cy2.getVolume());
   }
}
```

Example-11: Inheritance - The *Point2D* and *Point3D* Classes [3]



// Point2D.java

```
/*
 * The Point2D class models a 2D point at
(x, y).
 */
public class Point2D {
   // Private instance variables
   private int x, y;
```

// Point3D.java

```
/*
 * The Point3D class models a 3D point at
(x, y,z),
 * which is a subclass of Point2D.
 */
public class Point3D extends Point2D {
   // Private instance variables
   private int z;

   // Constructors
   public Point3D() {  // default
constructor
      super();     // x = y = 0
      this.z = 0;
   }
   public Point3D(int x, int y, int z) {
      super(x, y);
      this.z = z;
   }

   // Getters and Setters
   public int getZ() {
      return this.z;
   }
   public void setZ(int z) {
      this.z = z;
   }

   // Return "(x,y,z)"
   @Override
   public String toString() {
      return "(" + super.getX() + "," +
```

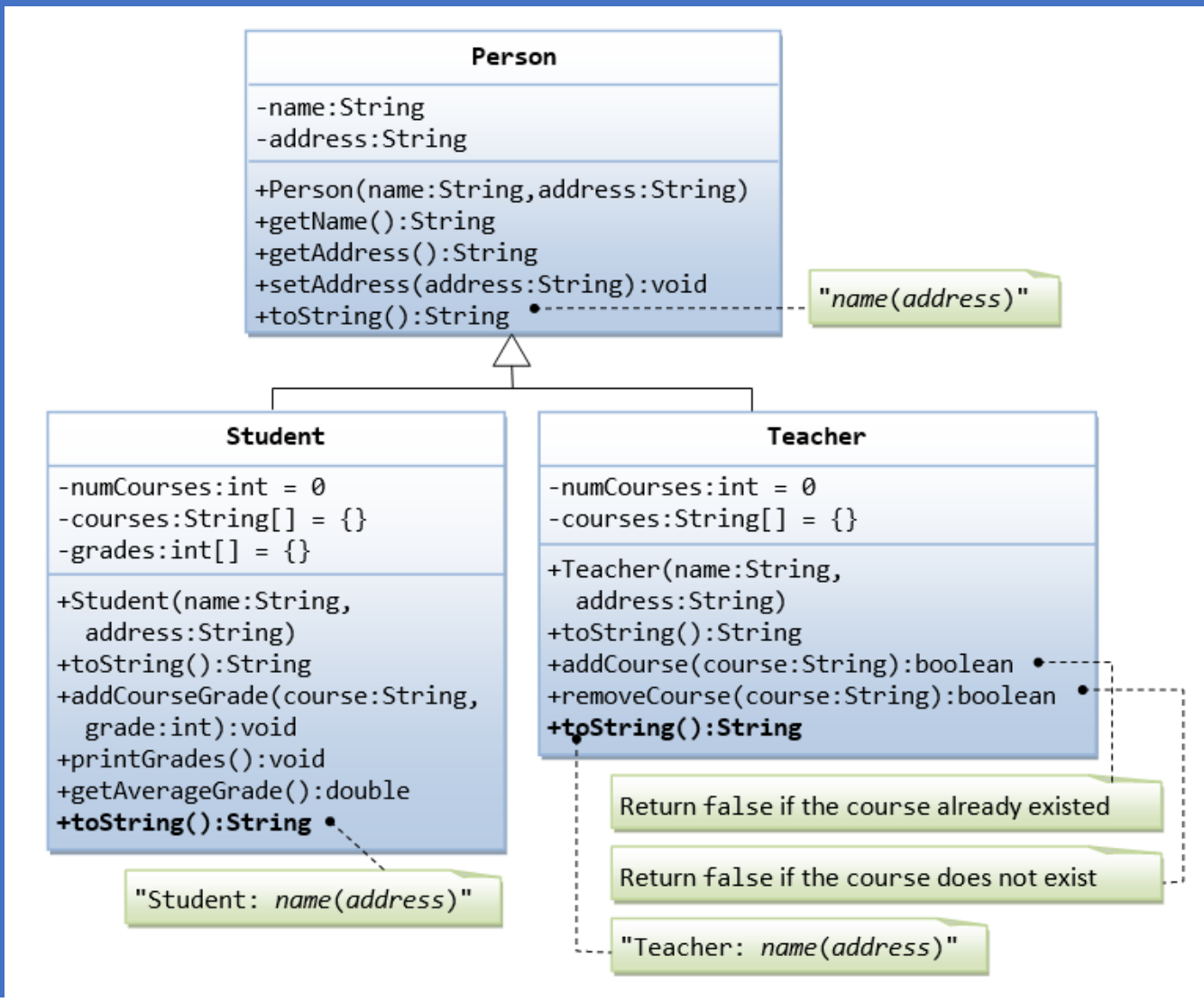
```
   // Constructors
   public Point2D() {  // default
constructor
      this.x = 0;
      this.y = 0;
   }
   public Point2D(int x, int y) {
      this.x = x;
      this.y = y;
   }

   // Getters and Setters
   public int getX() {
      return this.x;
   }
   public void setX(int x) {
      this.x = x;
   }
   public int getY() {
      return this.y;
   }
   public void setY(int y) {
      this.y = y;
   }

   // Return "(x,y)"
   public String toString() {
      return "(" + this.x + "," + this.y +
")";
   }
}
```

```
super.getY() + "," + this.z + ")";
   }
}
```

// TestPoint2DPoint3D.java

```
/*
 * A test driver for the Point2D and Point3D
classes
 */
public class TestPoint2DPoint3D {
   public static void main(String[] args) {
      /* Test Point2D */
      // Test constructors and toString()
      Point2D p2a = new Point2D(1, 2);
      System.out.println(p2a);  //
toString()
      Point2D p2b = new Point2D();  //
default constructor
      System.out.println(p2b);
      // Test Setters and Getters
      p2a.setX(3);  // Test setters
      p2a.setY(4);
      System.out.println(p2a);  //
toString()
      System.out.println("x is: " +
p2a.getX());
      System.out.println("x is: " +
p2a.getY());

      /* Test Point3D */
      // Test constructors and toString()
      Point3D p3a = new Point3D(11, 12, 13);
      System.out.println(p3a);  //
toString()
      Point2D p3b = new Point3D();  //
default constructor
      System.out.println(p3b);
      // Test Setters and Getters
      p3a.setX(21);  // in superclass
      p3a.setY(22);  // in superclass
      p3a.setZ(23);  // in this class
      System.out.println(p3a);  //
toString()
      System.out.println("x is: " +
p3a.getX());  // in superclass
      System.out.println("y is: " +
p3a.getY());  // in superclass
      System.out.println("z is: " +
p3a.getZ());  // in this class
   }
}
```

Example-12: Inheritance - Superclass *Person* and its Subclasses [3]



// Person.java

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

   // Constructor
   public Person(String name, String address) {
      this.name = name;
      this.address = address;
   }

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

   // Describle itself
   public String toString() {
      return name + "(" + address + ")";
   }
}
```

// 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

   // Constructor
   public Student(String name, String address)
{
      super(name, address);
      numCourses = 0;
      courses = new String[MAX_COURSES];
      grades = new int[MAX_COURSES];
   }

   // Describe itself
   @Override
   public String toString() {
      return "Student: " + super.toString();
   }
```

// 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

   // Constructor
   public Teacher(String name, String
address) {
      super(name, address);
      numCourses = 0;
      courses = new String[MAX_COURSES];
   }

   // Describe itself
   @Override
   public String toString() {
      return "Teacher: " +
super.toString();
   }

   // Return false if the course 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;
   }

   // Return 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-
```

```
   // Add a course and its grade - No
validation in this method
   public void addCourseGrade(String course,
int grade) {
      courses[numCourses] = course;
      grades[numCourses] = grade;
      ++numCourses;
   }

   // Print 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();
   }

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

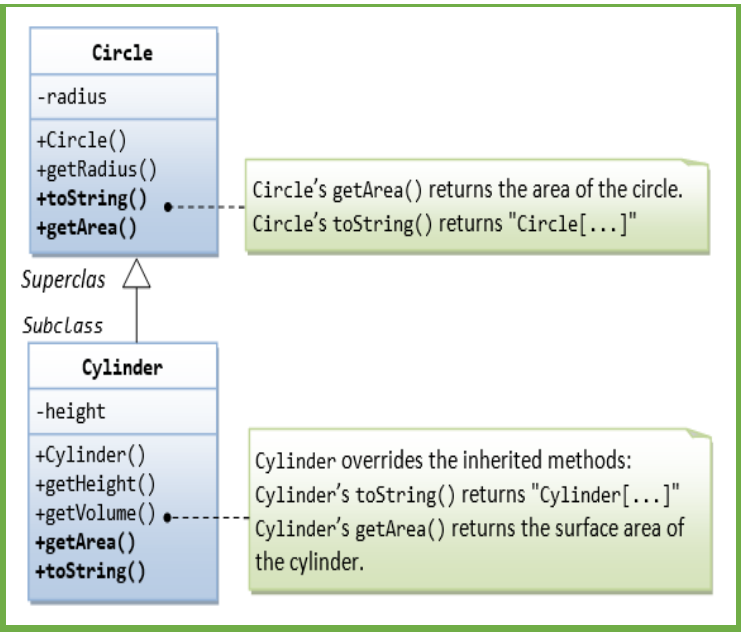
```
arrange for courses array
         for (int i = courseIndex; i <
numCourses-1; i++) {
            courses[i] = courses[i+1];
         }
         numCourses--;
         return true;
      } else {
         return false;
      }
   }
}
```

// 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.");
         }
      }
   }
}
```

Example-13: Polymorphism - *Circle* & *Cylinder* [3]

The word "*polymorphism*" means "*many forms*". It comes from Greek word "*poly*" (means *many*) and "*morphos*" (means *form*).



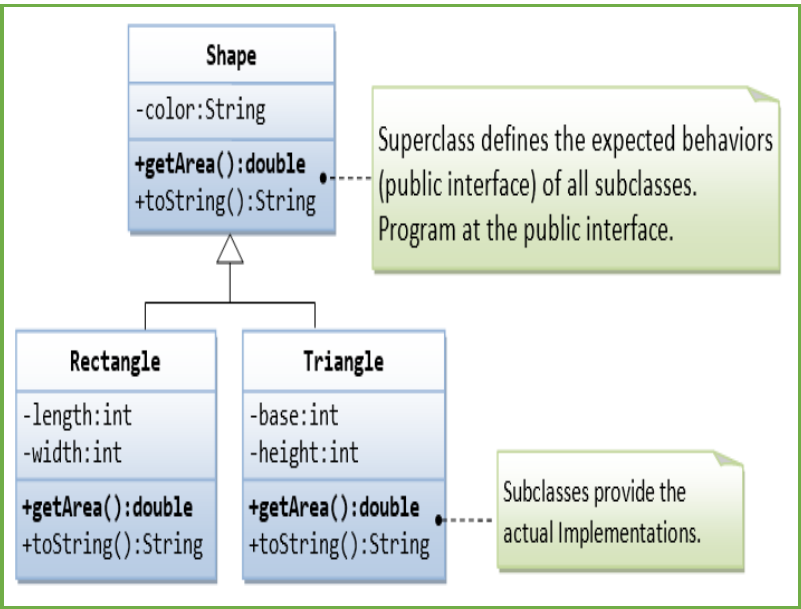
// Circle.java

```
// The superclass Circle
public class Circle {
   // private instance variable
   private double radius;
   // Constructor
   public Circle(double radius) {
      this.radius = radius;
   }
   // Getter
   public double getRadius() {
      return this.radius;
   }
   // Return the area of this circle
   public double getArea() {
      return radius * radius * Math.PI;
   }
   // Describe itself
   public String toString() {
      return "Circle[radius=" + radius + "]";
   }
}
```

// Cylinder.java

```
// The subclass Cylinder
public class Cylinder extends Circle {
   // private instance variable
   private double height;
   // Constructor
   public Cylinder(double height, double
radius) {
      super(radius);
      this.height = height;
   }
   // Getter
   public double getHeight() {
      return this.height;
   }
   // Return the volume of this cylinder
   public double getVolumne() {
      return super.getArea() * height;
   }
   // Override the inherited method to
return the surface area
   @Override
   public double getArea() {
      return 2.0 * Math.PI * getRadius() *
height;
   }
   // Override the inherited method to
describe itself
   @Override
   public String toString() {
      return "Cylinder[height=" + height +
"," + super.toString() + "]";
   }
}
```

Example-14: Polymorphism - *Shape* and its Subclasses [3]



// Shape.java

```
/*
 * Superclass Shape maintain the common properties of all shapes
 */
public class Shape {
   // Private member variable
   private String color;

   // Constructor
   public Shape (String color) {
      this.color = color;
   }

   @Override
   public String toString() {
      return "Shape[color=" + color + "]";
   }

   // All shapes must have a method called getArea().
   public double getArea() {
      // We have a problem here!
      // We need to return some value to compile the program.
      System.err.println("Shape unknown! Cannot compute area!");
      return 0;
   }
}
```

// Rectangle.java

```
/*
 * The Rectangle class, subclass of Shape
 */
public class Rectangle extends Shape {
   // Private member variables
   private int length;
   private int width;

   // Constructor
   public Rectangle(String color, int length, int width) {
      super(color);
      this.length = length;
      this.width = width;
   }

   @Override
   public String toString() {
      return "Rectangle[length=" + length + ",width=" + width + "," + super.toString() + "]";
   }

   // Override the inherited getArea() to provide the proper implementation
   @Override
   public double getArea() {
      return length*width;
   }
}
```

// Triangle.java

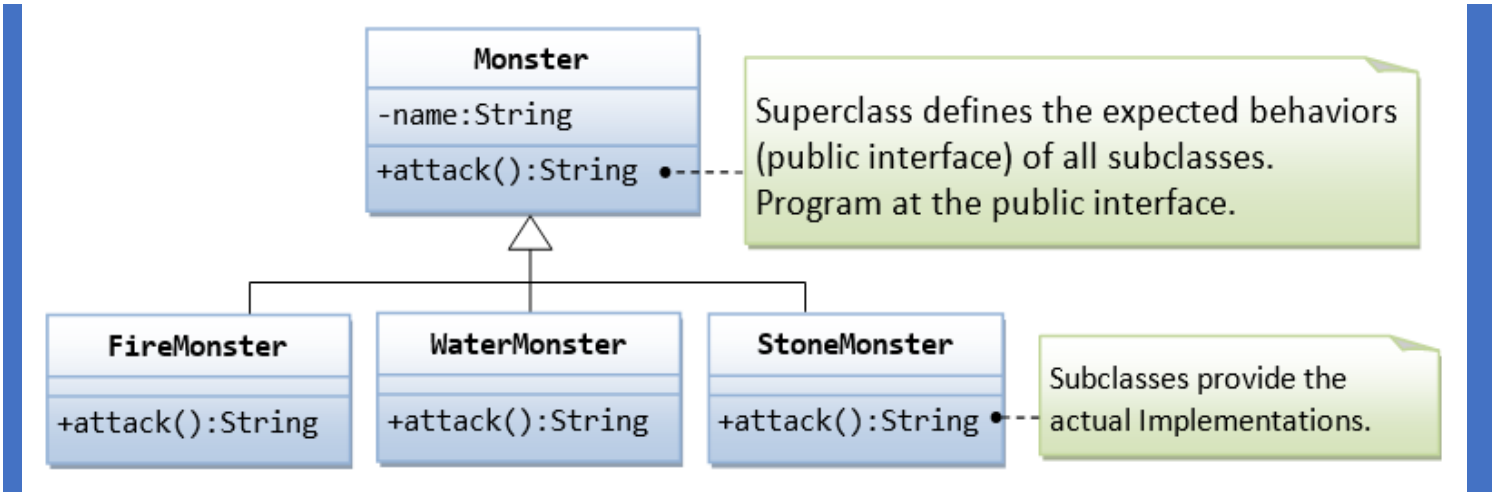
```
/*
 * The Triangle class, subclass of Shape
 */
public class Triangle extends Shape {
   // Private member variables
   private int base;
   private int height;

   // Constructor
   public Triangle(String color, int base, int height) {
      super(color);
      this.base = base;
      this.height = height;
   }

   @Override
   public String toString() {
      return "Triangle[base=" + base + ",height=" + height + "," + super.toString() + "]";
   }

   // Override the inherited getArea() to provide the proper implementation
   @Override
   public double getArea() {
      return 0.5*base*height;
   }
}
```

// TestShape.java

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

      Shape s2 = new Triangle("blue", 4, 5);  // Upcast
      System.out.println(s2);  // Run Triangle's toString()
      System.out.println("Area is " + s2.getArea());  // Run Triangle's getArea()
   }
```

```
   // Private member variables                    }
   private int length;
   private int width;

   // Constructor
   public Rectangle(String color, int length, int
width) {
      super(color);
      this.length = length;
      this.width = width;
   }

   @Override
   public String toString() {
      return "Rectangle[length=" + length +
",width=" + width + "," + super.toString() + "]";
   }

   // Override the inherited getArea() to provide
the proper implementation
   @Override
   public double getArea() {
      return length*width;
   }
}
```

Example -15: Polymorphism - *Monster* **and its Subclasses** [3]



// Monster.java
```
/*
 * The superclass Monster defines the
expected common behaviors for its
subclasses.
 */
public class Monster {
   // private instance variable
   private String name;

   // Constructor
   public Monster(String name) {
      this.name = name;
   }

   // Define common behavior for all its
subclasses
   public String attack() {
      return "!^_&^$@+%$* I don't know how
to attack!";
      // We have a problem here!
      // We need to return a String; else,
compilation error!
   }
}
```

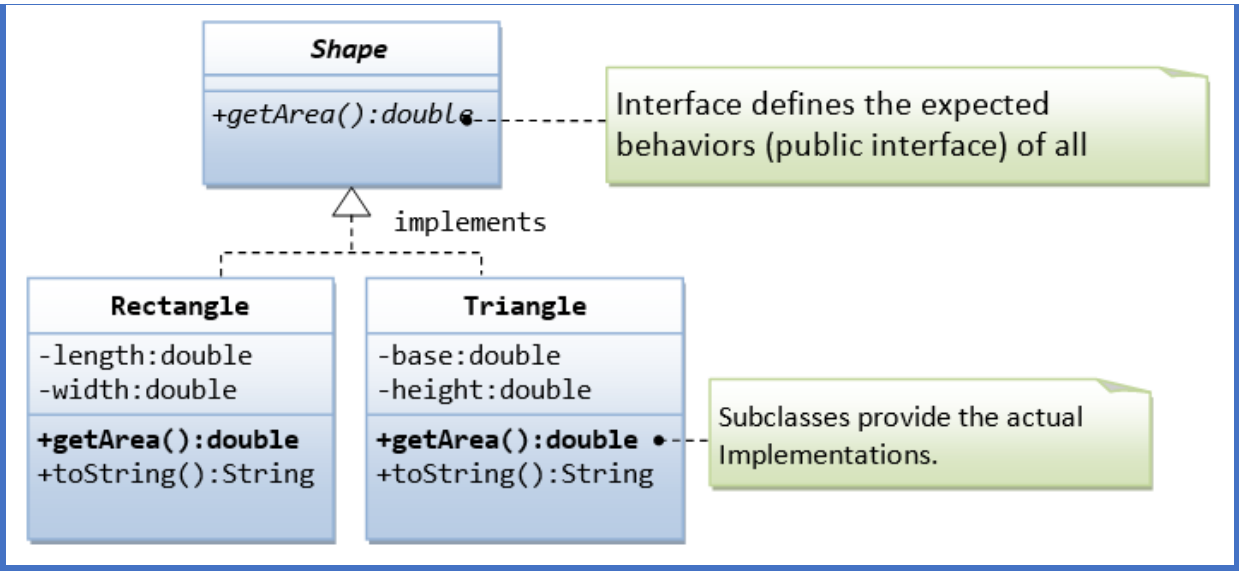
// FireMonster.java
```
public class FireMonster extends Monster {
   // Constructor
   public FireMonster(String name) {
      super(name);
   }
   // Subclass provides actual
implementation
   @Override public String attack() {
      return "Attack with fire!";
   }
}
```

// WaterMonster.java
```
public class WaterMonster extends Monster {
   // Constructor
   public WaterMonster(String name) {
      super(name);
   }
   // Subclass provides actual
implementation
   @Override public String attack() {
      return "Attack with water!";
   }
}
```

// StoneMonster.java
```
public class StoneMonster extends Monster {
   // Constructor
   public StoneMonster(String name) {
      super(name);
   }
   // Subclass provides actual
implementation
   @Override public String attack() {
      return "Attack with stones!";
   }
}
```

// TestMonster.java
```
public class TestMonster {
   public static void main(String[] args) {
      // Program at the "interface" defined
in the superclass.
      // Declare instances of the
superclass, substituted by subclasses.
      Monster m1 = new FireMonster("r2u2");
// upcast
      Monster m2 = new WaterMonster("u2r2");
// upcast
      Monster m3 = new StoneMonster("r2r2");
// upcast

      // Invoke the actual implementation
      System.out.println(m1.attack());  //
Run FireMonster's attack()
      System.out.println(m2.attack());  //
Run WaterMonster's attack()
      System.out.println(m3.attack());  //
Run StoneMonster's attack()

      // m1 dies, generate a new instance
and re-assign to m1.
      m1 = new StoneMonster("a2b2");  //
upcast
      System.out.println(m1.attack());  //
Run StoneMonster's attack()

      // We have a problem here!!!
      Monster m4 = new Monster("u2u2");
      System.out.println(m4.attack());  //
garbage!!!
   }
}
```

Example-16: Interface - *Shape* Interface and its Implementations [3]



UML Notations: **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.**

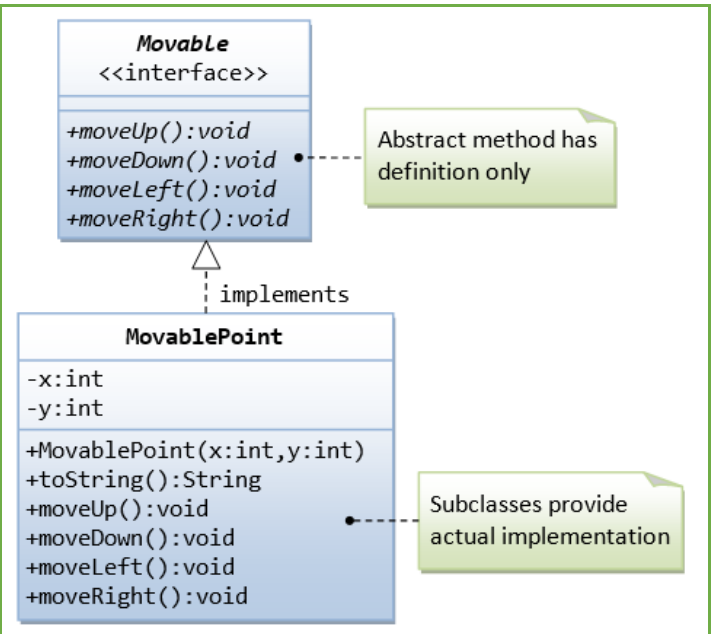
```
/*
 * The interface Shape specifies the behaviors
 * of this implementations subclasses.
 */
public interface Shape {  // Use keyword
"interface" instead of "class"
   // List of public abstract methods to be
implemented by its subclasses
   // All methods in interface are "public
abstract".
   // "protected", "private" and "package"
methods are NOT allowed.
   double getArea();
}

// 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;
   private int width;

   // Constructor
   public Rectangle(int length, int width) {
      this.length = length;
      this.width = width;
   }

   @Override
   public String toString() {
      return "Rectangle[length=" + length +
",width=" + width + "]";
   }

   // Need to implement all the abstract
```

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

   // Constructor
   public Triangle(int base, int height) {
      this.base = base;
      this.height = height;
   }

   @Override
   public String toString() {
      return "Triangle[base=" + base +
",height=" + height + "]";
   }

   // Need to implement all the abstract
methods defined in the interface
   @Override
   public double getArea() {
      return 0.5 * base * height;
   }
}


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

      Shape s2 = new Triangle(3, 4);  //
```

```
methods defined in the interface
   @Override
   public double getArea() {
      return length * width;
   }
}
```

```
upcast
      System.out.println(s2);
      System.out.println("Area is " +
s2.getArea());

      // Cannot create instance of an
interface
      //Shape s3 = new Shape("green");   //
Compilation Error!!
   }
}
```

Example-17: Interface - *Movable* Interface and its Implementations [3]



// Moveable.java

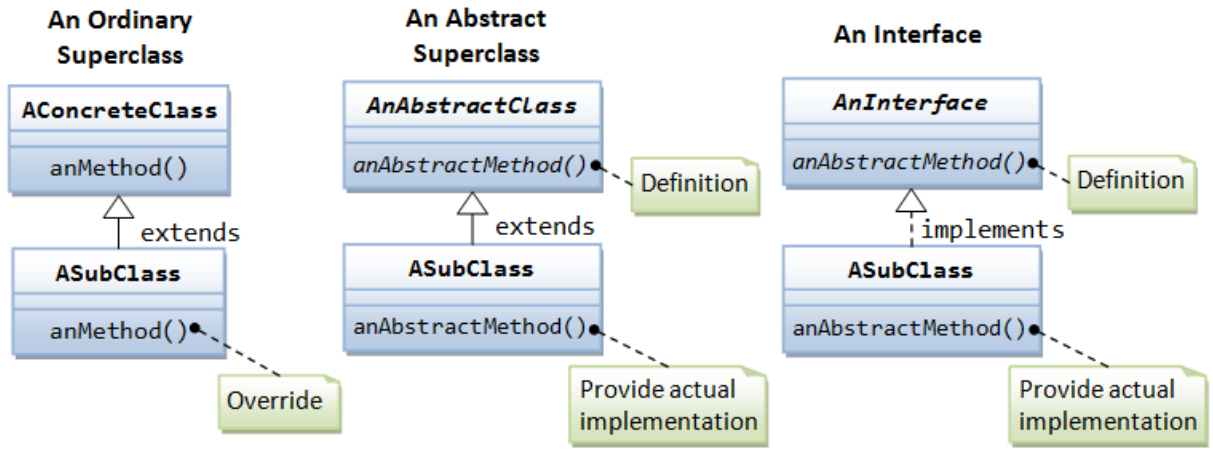
```
/*
 * The Movable interface defines a list of
public abstract methods
 * to be implemented by its subclasses
 */

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 the
subclasses
   public void moveUp();    // "public" and
"abstract" optional
   public void moveDown();
   public void moveLeft();
   public void moveRight();
}
```

// MovablePoint.java

```
// The subclass MovablePoint needs to
implement all the abstract methods
// defined in the interface Movable

public class MovablePoint implements Movable
{
   // Private member variables
   private int x, y;   // (x, y) coordinates
of the point

   // Constructor
   public MovablePoint(int x, int y) {
      this.x = x;
      this.y = y;
   }

   @Override
   public String toString() {
      return "(" + x + "," + y + ")";
   }

   // Need to implement all the abstract
methods defined in the interface Movable
   @Override
   public void moveUp() {
      y--;
   }
   @Override
   public void moveDown() {
      y++;
   }
   @Override
   public void moveLeft() {
      x--;
   }
   @Override
   public void moveRight() {
      x++;
   }
}
```

```
// TestMovable.java
public class TestMovable {
   public static void main(String[] args) {
      MovablePoint p1 = new MovablePoint(1,
2);  // upcast
      System.out.println(p1);
      p1.moveDown();
      System.out.println(p1);
      p1.moveRight();
      System.out.println(p1);

      // Test Polymorphism
      Movable p2 = new MovablePoint(3, 4);
// upcast
      p2.moveUp();
      System.out.println(p2);
      MovablePoint p3 = (MovablePoint)p2;
// downcast
      System.out.println(p3);
   }
}
```

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.



Why interfaces?

An interface is a *contract* (or a protocol, or a common understanding) of what the classes can do. When a class implements a certain interface, it promises to provide implementation to all the abstract methods declared in the interface. Interface defines a set of common behaviors. The classes implement the interface agree to these behaviors and provide their own implementation to the behaviors. This allows you to program at the interface, instead of the actual implementation. One of the main usage of interface is provide a *communication contract* between two objects. If you know a class implements an interface, then you know that class contains concrete implementations of the methods declared in that interface, and you are guaranteed to be able to invoke these methods safely. In other words, two objects can communicate based on the contract defined in the interface, instead of their specific implementation.

Interface vs. Abstract Superclass

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

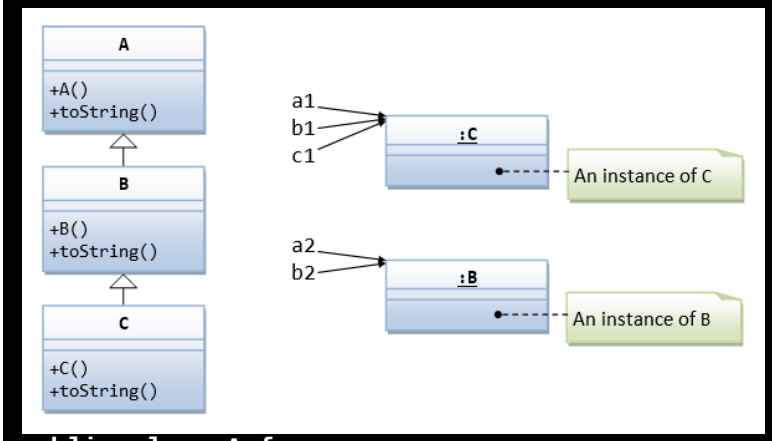
Use abstract superclass if there is a clear class hierarchy. Abstract class can contain partial implementation (such as instance variables and methods). Interface cannot contain any implementation, but merely defines the behaviors.

Upcasting & Downcasting

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,

```
Circle c1 = new Cylinder(1.1, 2.2);  // Compiler checks to ensure that R-value is a subclass of L-value.
Circle c2 = new String();            // Compilation error: incompatible types
```

Example on Upcasting and Downcasting




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

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

public class C extends B {
   public C() {  // Constructor
      super();
      System.out.println("Constructed 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
      C c1 = (C)b1;     // downcast okay

      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 ClassCastException
   }
}
```

Using Interfaces as Abstract Base Classes [4]

Given inheritance hierarchy, shown graphically in Figure-1 below, you could not instantiate a `Cup` object, but you could use a `Cup` reference to send messages to a `CoffeeCup` or `TeaCup` object. Given a `Cup` reference that refers to a `CoffeeCup` or `TeaCup`, you could invoke `add()`, `releaseOneSip()`, or `spillEntireContents()` on it. The implementation of those methods that actually gets invoked at run-time will depend upon the actual class of the object referred to by the `Cup` reference.

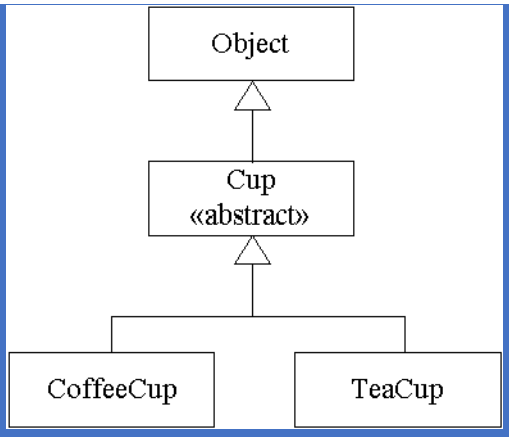


Figure-1: `Cup` as an abstract base class

Since this class `Cup` contains only public abstract methods, it could alternatively be declared as an interface:

```
// Cup.java
interface Cup {
    void add(int amount);
    int removeOneSip(int sipSize);
    int spillEntireContents();
}

// CoffeeCup.java
class CoffeeCup implements Cup {
    public void add(int amount) {
        //...
    }
    public int removeOneSip(int sipSize) {
        //...
        return 0;
    }
    public int spillEntireContents() {
        //...
        return 0;
    }
    //...
}

// TeaCup.java
class TeaCup implements Cup {
    public void add(int amount) {
        //...
    }
    public int removeOneSip(int sipSize) {
```

Here you are using an interface to represent an abstract base class. As suggested by the Java Language Specification, the name of the class, `Cup`, is a noun. The inheritance hierarchy for this is shown in Figure-2.

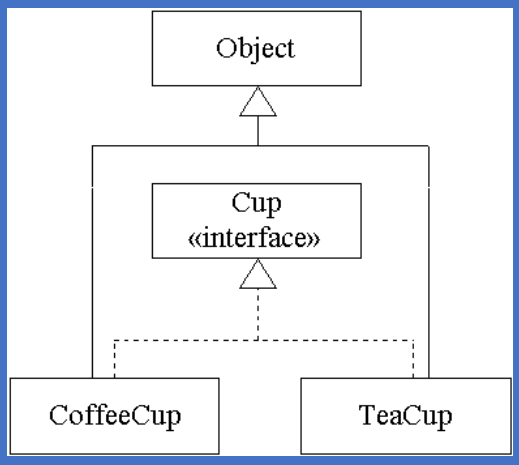
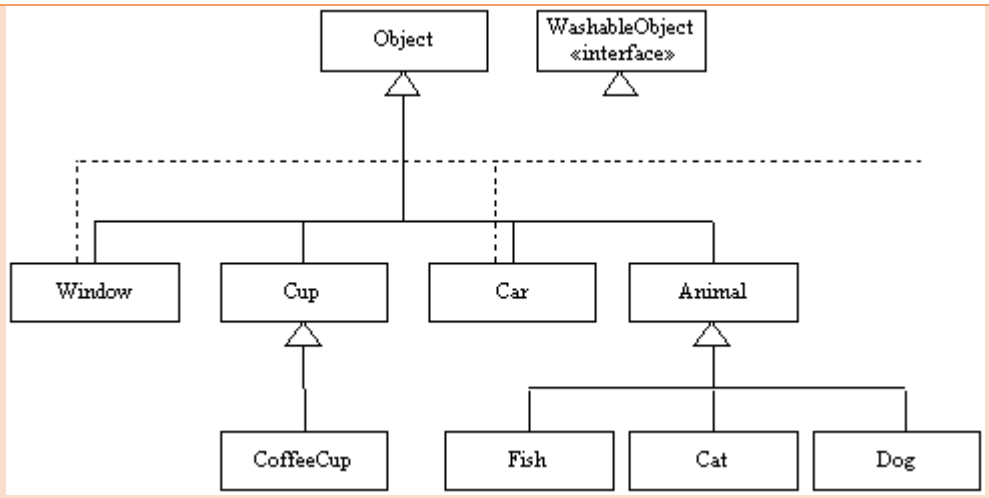


Figure 2. `Cup` as an interface

In general, if you have an abstract base class that declares only public abstract methods and public static final fields, you may as well make it an interface. Because an abstract class is restricted to single inheritance, but an interface can be multiply inherited, an interface is more flexible than an abstract class.

```
        //...
        return 0;
    }
    public int spillEntireContents() {
        //...
        return 0;
    }
    //...
}
```

The Interface Solution [4]




```
// Washable.java
interface Washable {
    void wash();
}

// Window.java
class Window implements Washable {
    public void wash() {
        System.out.println("Washing a
Window.");
        //...
    }
    //...
}

// Cup.java
class Cup implements Washable {
    public void wash() {
        System.out.println("Washing a
Cup.");
        // ...
    }
    //...
}

// CoffeeCup.java
class CoffeeCup extends Cup {
    public void wash() {
        System.out.println("Washing a
CoffeeCup.");
        // ...
```

```
// Animal.java
class Animal {
    //...
}


// Dog.java
class Dog extends Animal implements Washable
{
    public void wash() {
        System.out.println("Washing a
Dog.");
        //...
    }
    //...
}

// Cat.java
class Cat extends Animal {
    //...
}

// Fish.java
class Fish extends Animal {
    //...
}
```

```
    }
    //...
}

// CoffeeMug.java
class CoffeeMug extends CoffeeCup {
    public void wash() {
        System.out.println("Washing a
CoffeeMug.");
        // ...
    }
    //...
}

// EspressoCup.java
class EspressoCup extends CoffeeCup {
    public void wash() {
        System.out.println("Washing an
EspressoCup.");
        // ...
    }
    //...
}

// Car.java
class Car implements Washable {
    public void wash() {
        System.out.println("Washing a
Car.");
        //...
    }
    //...
}
```

The inheritance hierarchy for these classes is shown in Figure above. Given these definitions for cups, animals, windows, and cars, you could once again get the benefits of polymorphism when writing method `cleanAnObject()`:

```
// Cleaner.java
class Cleaner {
    public static void
cleanAnObject(Washable washMe) {
        //...
        washMe.wash();
        //...
    }
}
```

Using `instanceof` with Interfaces [4]

Given a reference to an object, you can find out if a particular interface is a superinterface of that object's class by using `instanceof`. For example, the `washIfPossible()` method, shown below, uses `instanceof` to determine whether an object is a subtype of the `Washable` interface:

```
// In Source Packet in file interface/ex9/Example9.java
class Example9 {

    public static void washIfPossible(Object o) {
        if (o instanceof Washable) {
            // Washable is a superinterface of the
            // object's class
            ((Washable) o).wash();
        }
        else {
            System.out.println("Can't wash this.");
        }
    }

    public static void main(String[] args) {
        washIfPossible(new Cup());
        washIfPossible(new CoffeeCup());
        washIfPossible(new CoffeeMug());
```

```
        washIfPossible(new EspressoCup());
        washIfPossible(new Car());
        washIfPossible(new Animal());
        washIfPossible(new Dog());
        washIfPossible(new Cat());
        washIfPossible(new Fish());
        washIfPossible(new Window());
    }
}
```

NOTE

https://www.visual-paradigm.com/tutorials/netbeans-java-to-uml.jsp >>> **How to Draw UML Diagrams in NetBeans?**

How to Generate Java from UML Class Diagram in NetBeans?

References:

[1] https://www.tutorialspoint.com/object_oriented_analysis_design/ooad_uml_structural_diagrams.htm

[2] http://www.cs.sjsu.edu/~pearce/modules/lectures/oop/basics/interfaces.htm

[3] https://www.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html

[4] https://www.artima.com/objectsandjava/webuscript/PolymorphismInterfaces1.html