UML

Basic UML for Class Diagram

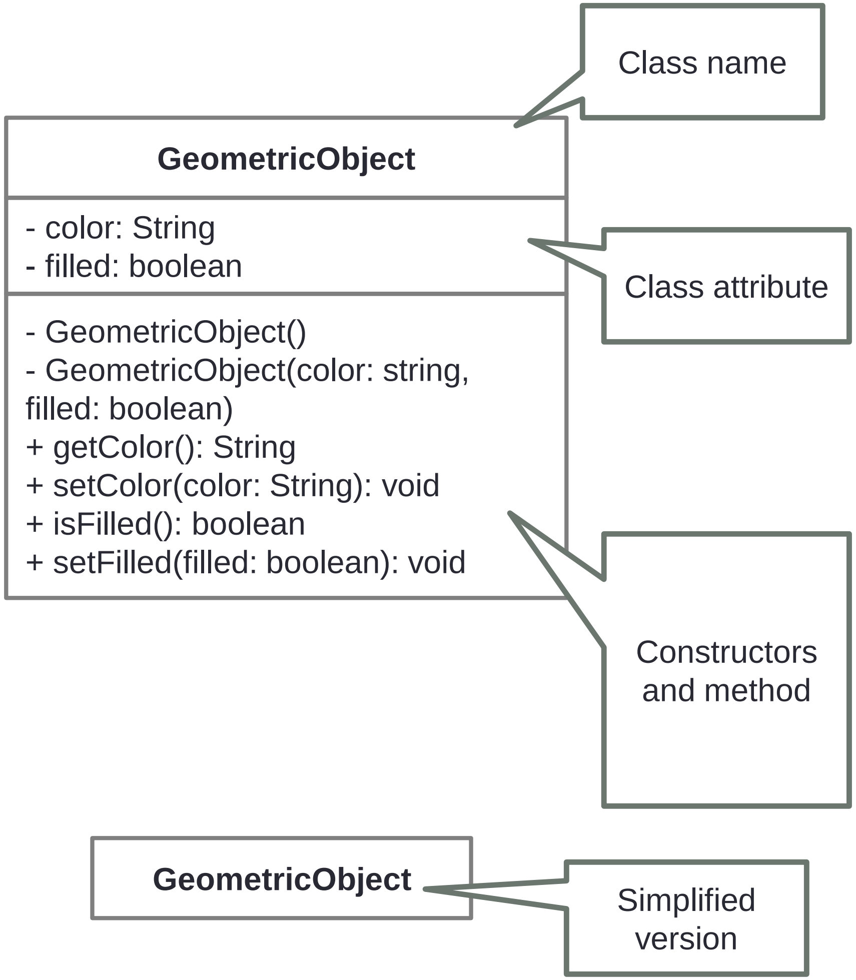
- http://www.uml.org/
- http://www.classdraw.com/help.htm
- https://nirajrules.wordpress.com/2011/07/15/association-vs-dependency-vs-aggregation-vs-composition/
- sci.feu.ac.th/boonrit/ood/class%20**relation**.ppt

UML Diagrams

- UML stands for the *Unified Modeling Language*
- *UML diagrams* show relationships among classes and objects
- A UML *class diagram* consists of one or more classes, each with sections for the class name, attributes (data), and operations (methods)
- Lines between classes represent *associations*
- A dotted arrow shows that one class *uses* the other (calls its methods)

UML Symbol

- Class
  - Class name is defined at the top of the class symbol
  - Attributes is defined at the second part together with their types
  - Constructors and methods are listed at the last part together with their parameters and return types
  - Sign denote the visibility modifier
    - - denotes private
    - # denotes protected
    - + denotes public



Class name

GeometricObject

- color: String
- filled: boolean

Class attribute

- GeometricObject()
- GeometricObject(color: string, filled: boolean)
+ getColor(): String
+ setColor(color: String): void
+ isFilled(): boolean
+ setFilled(filled: boolean): void

Constructors and method

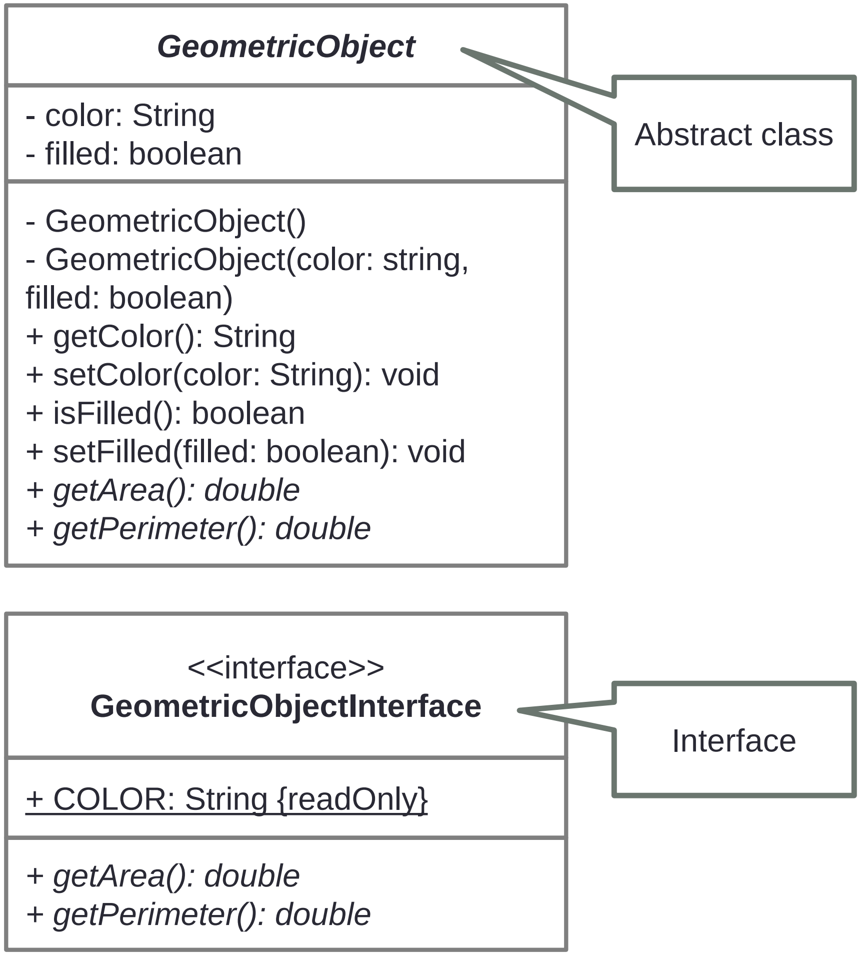
GeometricObject

Simplified version

UML Symbol

- Class
  - Abstract class and abstract methods are italicized.

    Sometimes you can see stereotype of abstract.
  - Interface is denoted by using stereotype.
  - Static variable is denoted by underlined text



GeometricObject

- color: String
- filled: boolean

- GeometricObject()
- GeometricObject(color: string, filled: boolean)
+ getColor(): String
+ setColor(color: String): void
+ isFilled(): boolean
+ setFilled(filled: boolean): void
+ *getArea(): double*
+ *getPerimeter(): double*

Abstract class

<<interface>>
GeometricObjectInterface

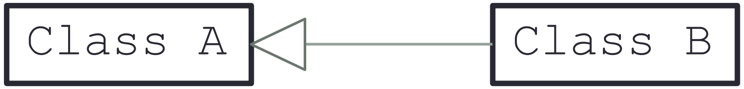
+ COLOR: String {readOnly}

+ *getArea(): double*
+ *getPerimeter(): double*

Interface

UML Symbol

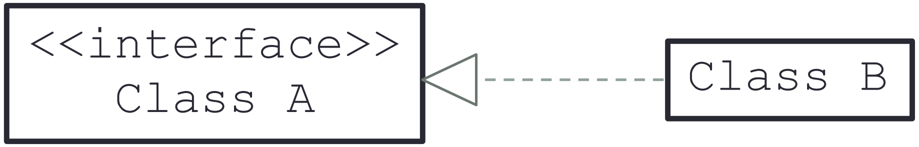
- Inheritance (A is a super class, B is a sub class)



Class A

Class B

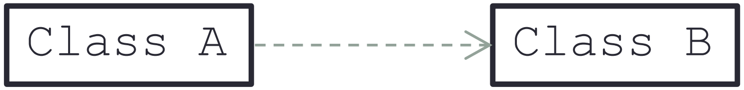
- Interface



<<interface>>
Class A

Class B

- Dependency

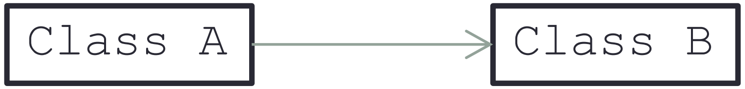


Class A

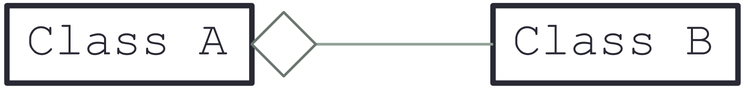
Class B

UML Symbol

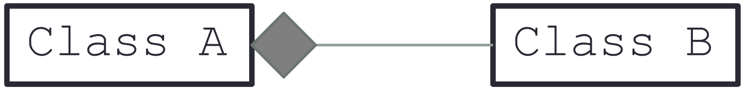
- Association (Class A holds a class reference to Class B)



- Aggregation

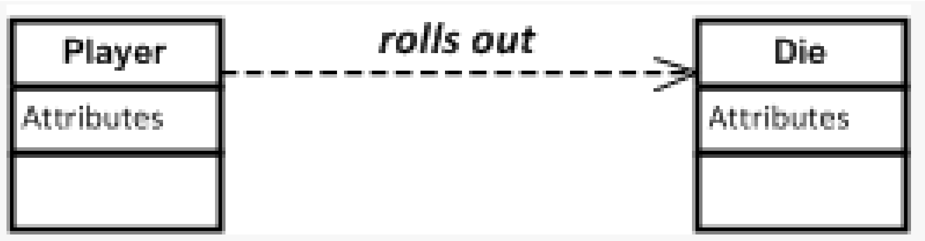


- Composition



UML Symbol

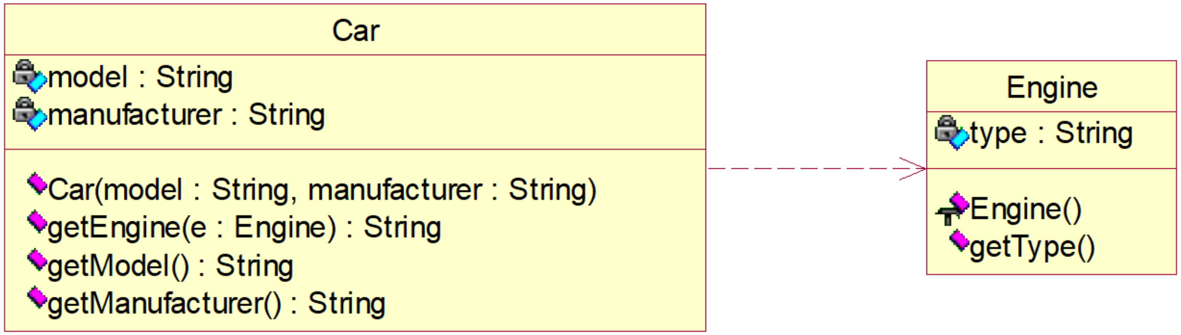
- Dependency
  - Normally created when you receive a reference to a class as part of a particular operation / method.



```
class Die {
   public void Roll() { ... }
}

class Player
{
public void TakeTurn(Die die) /*Look, I’m dependent on Die and
it's Roll method to do my work*/
{
   die.Roll(); ... }
}
```

UML Symbol

- Dependency

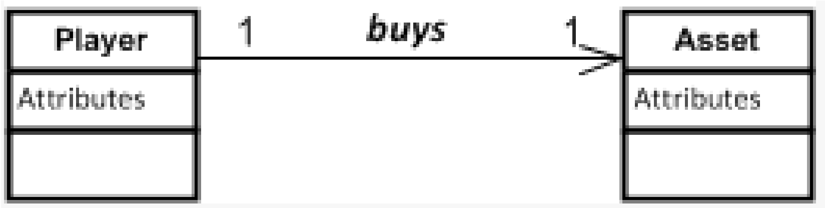

Car
model : String
manufacturer : String
Car(model : String, manufacturer : String)
getEngine(e : Engine) : String
getModel() : String
getManufacturer() : String
Engine
type : String
Engine()
getType()

```
class Car {
  private String model;
  private String manufacturer;
  public Car (String model, String manufacturer){
         this.model =model;
         this.manufacturer =manufacturer;
  }
  public String getEngine (Engine e) {

         return e.getType();
  }
  public String getModel(){
         return model;
  }
  public String getManufacturer(){
         return manufacturer;
  }
}
```

```
class Engine {
  private String type;
  Engine (String type){
         this.type =type;
  }
  public String getType(){
         return type;
  }
}
```

UML Symbol

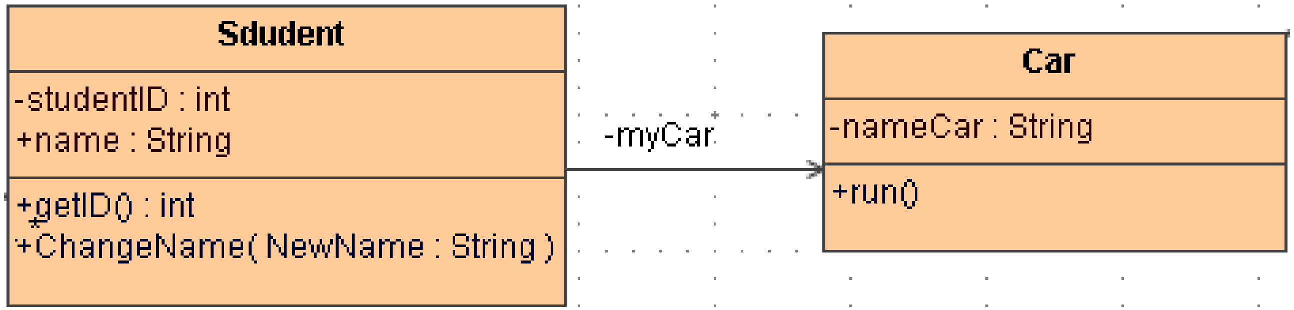
- Association
  - Association defines the multiplicity between objects.



```
class Asset { ... }

class Player {
   Asset asset;
   public Player(Asset purchasedAsset) { ... } /*Set the asset via
Constructor or a setter*/
}
```

UML Symbol

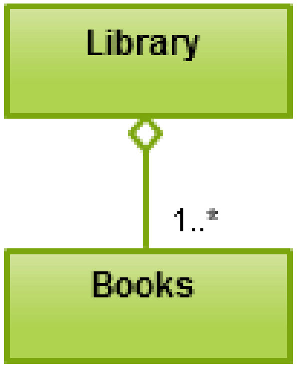
- Association



```
public class Sdudent
{
        private int studentID;

        public String name;

        private Car myCar;

        public int getID( )
        {
                return 0;
        }

        public void ChangeName( final String NewName )
        {

        }

}
```

UML Symbol

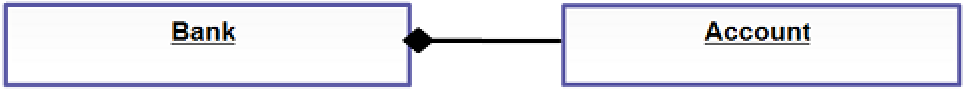
- Aggregation
  - Aggregation is a special type of association. In aggregation, objects have their own life cycle but there is an ownership. Whenever we have “HAS-A” relationship between objects and ownership then it’s a case of aggregation.



Aggregation

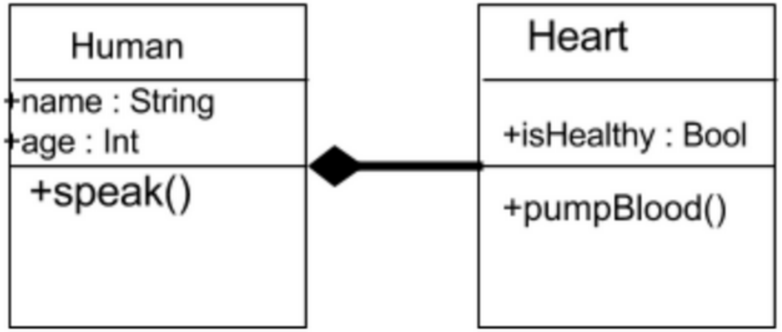
UML Symbol

- Composition
  - Composition is a special case of aggregation. Composition is a more restrictive form of aggregation. When the contained object in “HAS-A” relationship can’t exist on it’s own, then it’s a case of composition. For example, House has-a Room. Here room can’t exist without house.



Bank

Account



Human

+name : String
+age : Int

+speak()

Heart

+isHealthy : Bool

+pumpBlood()

Recursion

References:

- Y. Daniel Liang, ***“Introduction to Java Programming, Comprehensive Version”***, Pearson Education Inc., **2012 (9th edition)**
- **https://www.cs.utah.edu/~germain/PPS/Topics/recursion.html**

Recursion

- Recursion means "defining a problem in terms of itself".

- In general, to solve a problem using recursion, you break it into subproblems. If a subproblem resembles the original problem, you can apply the same approach to solve the subproblem recursively. This subproblem is almost the same as the original problem in nature with a smaller size.

- Recursion solves such recursive problems by using functions that call themselves

Recursion

- All recursive methods have the following characteristics:
  - One or more base cases (the simplest case) are used to stop recursion.
  - Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

Example - Computing Factorial

- factorial(0) = 1;
- factorial(n) = n*factorial(n-1);

- n! = n * (n-1)!

Example - Computing Factorial

- factorial(3) = 3 * factorial(2)
  = 3 * (2 * factorial(1))
  = 3 * (2 * (1 * factorial(0)))
  = 3 * (2 * (1 * 1)))
  = 3 * (2 * 1)
  = 3 * 2
  = 6

- Therefore: factorial(n) = n*factorial(n-1) with the base case factorial(0) = 1

Example - Computing Factorial

```
public class FactorialRecursion {
    public static void main(String[] args) {
        FactorialRecursion fact = new FactorialRecursion();
        System.out.println(fact.factorial(5));
    }
    public int factorial(int n){
        if(n==0){
            return 1;
        }
        else{
            return n * factorial(n-1);
        }
    }
}
```

Exercise

- Write a program to compute the Fibonacci number at the nth position by using recursion.