Object-Oriented Concepts Review

Overview

- Abstract Data Type
- Object-Oriented Paradigm
- Inheritance
- Polymorphism
  - Method Overloading
  - Method Overriding
- What is an Interface?
- Polymorphism Examples
- Class Relationships and Change Propagations

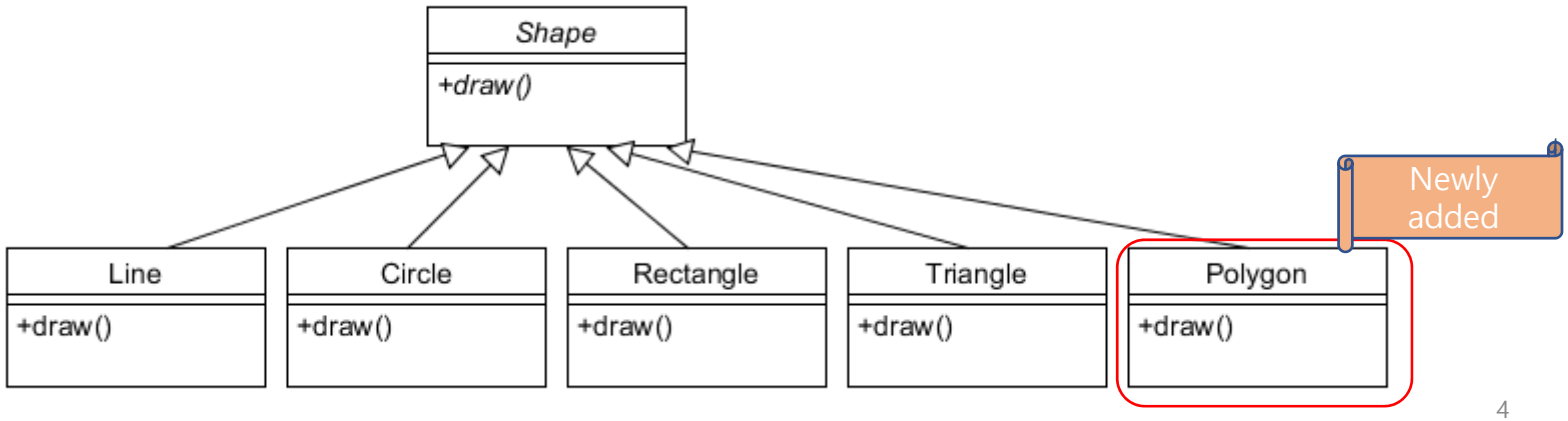
Introduction to Abstract Data Type (ADT)

- An *abstract data type* is a user-defined data type that satisfies the following two conditions:
  - The representation of, and operations on, objects of the type are defined in a **single syntactic unit**
    - Increases program **organization**, **modifiability** (everything associated with a data structure is together), and **separate compilation**
  - The **representation** of objects of the type is **hidden** from the program units that use these objects, so the only operations possible are those provided in the type's definition
    - Increases **reliability;** by hiding the data representations, user code cannot directly access objects of the type or depend on the representation, **allowing the representation to be changed without affecting user code**

Object-Oriented Paradigm

Class = ADT + Inheritance + Polymorphism

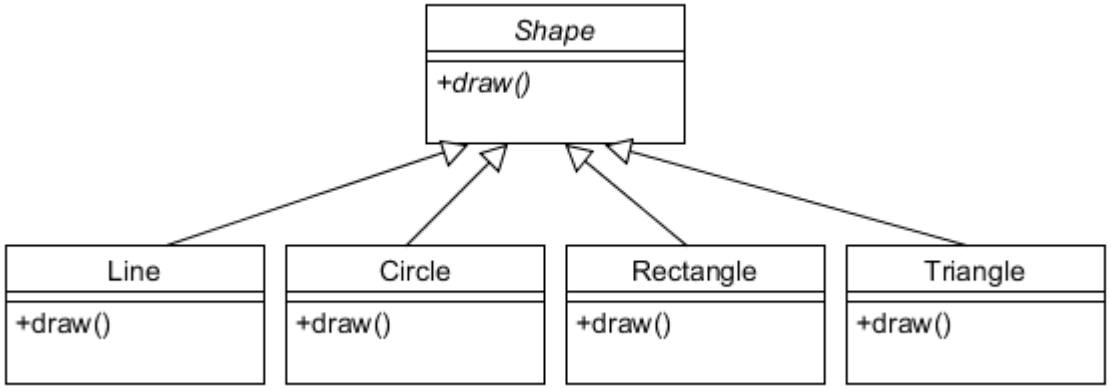
Promotes reusability and flexibility



Object-Oriented Paradigm

Class = ADT + Inheritance + Polymorphism

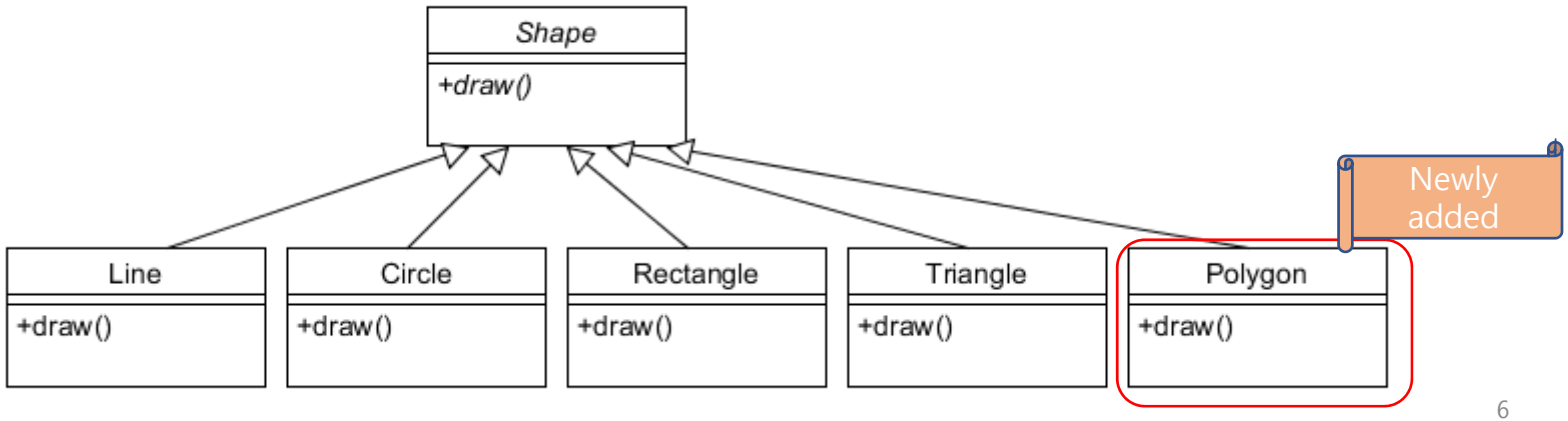
```
class ClientOfShape
{
   void renderShape(Shape s)
   {
       if (s.type == LINE) DrawLine(s.data);
       else if ((s.type == CIRCLE) DrawCirlce(s.data);
       else if ((s.type == RECT) DrawRect(s.data);
       else if ((s.type == TRIANGLE) DrawTriangle(s.data);
   }
}
```



Object-Oriented Paradigm

Class = ADT + Inheritance + Polymorphism

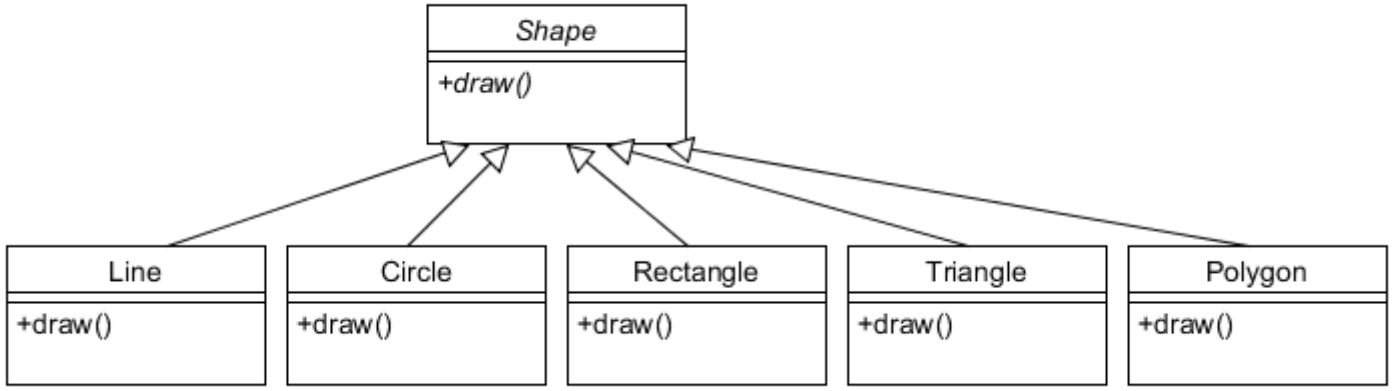
```
class ClientOfShape
{
   void renderShape(Shape s)
   {
       if (s.type == LINE) DrawLine(s.data);
       else if ((s.type == CIRCLE) DrawCirlce(s.data);
       else if ((s.type == RECT) DrawRect(s.data);
       else if ((s.type == TRIANGLE) DrawTriangle(s.data);
       else if ((s.type == POLYGON) DrawPolygon(s.data);
   }
}
```



Object-Oriented Paradigm

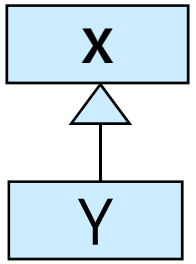
Class = ADT + Inheritance + Polymorphism

```
class ClientOfShape
{
   void renderShape(Shape s)
   {
       s.draw(); // No Change!
   }
}
```



Inheritance

- When class Y inherits from class X
  - Y inherits all the methods of X
    - Instances of Y can be sent all the messages that X responds to
  - Y inherits all the data from X
    - Instances Y have instances of all the data of X
  - As a consequence we can say
    - Y is a X
  - Every instance of Y is also an instance of X
  - **We can use the instance of Y wherever the instance of type X is expected**



Polymorphism

- A property of object oriented software by which an *operation (typically virtual) may be performed in different ways* in different classes.
  - Requires that there be *multiple methods of the same name*
  - The choice of which one to execute depends on the object that is in a variable
  - Reduces the need for programmers to code many if-else or switch statements
- Types of Polymorphism
  - Runtime polymorphism (Dynamic polymorphism)
    - Method Overriding
  - Compile time polymorphism (Static polymorphism)
    - Method Overloading

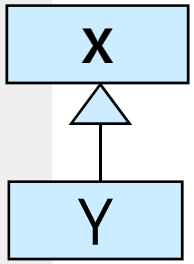
Method Overloading

```
class X
{
   void methodA(int num)
   {
       System.out.println ("methodA:" + num);
   }
   void methodA(int num1, int num2)
   {
       System.out.println ("methodA:" + num1 + "," + num2);
   }
   double methodA(double num) {
       System.out.println("methodA:" + num);
       return num;
   }
}

public class test
{
   public static void main (String args [])
   {
       X Obj = new X();
       double result;
       Obj.methodA(20);
       Obj.methodA(20, 30);
       result = Obj.methodA(5.5);
       System.out.println("Answer is:" + result);
   }
}
```

```
methodA:20
methodA:20,30
methodA:5.5
Answer is:5.5
```

Method Overriding

```
public class X
{
    public void methodA() //Base class method
    {
        System.out.println ("hello, I'm methodA of class X");
    }
}

public class Y extends X
{
    public void methodA() //Derived Class method
    {
        System.out.println ("hello, I'm methodA of class Y");
    }
}
public class Z
{
   public static void main (String args []) {
       X obj1 = new X(); // Reference and object X
       X obj2 = new Y(); // X reference but Y object
       obj1.methodA();
       obj2.methodA();
   }
}
```




```
hello, I'm methodA of class X
hello, I'm methodA of class Y
```

(Run-time) Polymorphism

- A *polymorphic variable* can be defined in a class that is able to **reference** (or point to) **objects of the class** and **objects of any of its descendants**

- When a class hierarchy includes classes that override methods and such methods are called through a polymorphic variable, the binding to the **correct method** will be dynamic

- Allows software systems to be **more easily extended during both development and maintenance**

Abstract Class and Abstract Method (in Java)

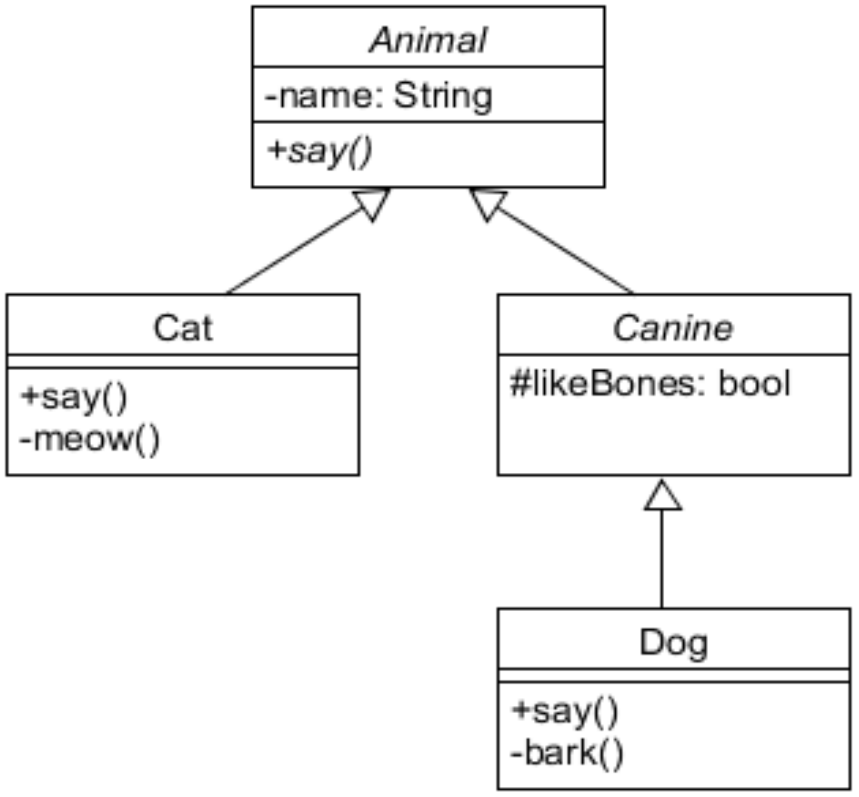
- An *abstract method* is one that does not include a definition (it only defines a protocol)
- A class defined with “abstract” keyword is an abstract class
- A class with at least one abstract method should become an abstract class (but not vice versa)
- An abstract class cannot be instantiated
- But, a variable of abstract class type can refer to any objects of its decedents!

Polymorphism + Abstract class

```
abstract class Animal {
    protected String name;
    abstract public void say();
}

class Cat extends Animal {
    private void meow() { ... }
    public void say() { meow(); }
}

abstract class Canine extends Animal {
    protected bool likeBones;
}

class Dog extends Canine {
    private void bark() { ... }
    public void say() { bark(); }
}
```

```
Animal  a = null;
Dog  baduki = new Dog();
Cat  nabi = new Cat();

a = baduki; a.say();
a = nabi;  a.say();

Animal c1 = new Animal();
Animal c2 = new Cat();
Cat c3 = new Animal();
Cat c4 = new Cat();

Animal d1 = new Canine();
Animal d2 = new Dog();
Canine d3 = new Dog();
Canine d4 = new Cat();
Dog d5 = new Canine();
Dog d6 = new Dog();
```

Polymorphism + Abstract class




```
Animal  a = null;
Dog  baduki = new Dog();
Cat  nabi = new Cat();

a = baduki; a.say();
a = nabi;  a.say();

Animal c1 = new Animal();
Animal c2 = new Cat();
Cat c3 = new Animal();
Cat c4 = new Cat();

Animal d1 = new Canine();
Animal d2 = new Dog();
Canine d3 = new Dog();
Canine d4 = new Cat();
Dog d5 = new Canine();
Dog d6 = new Dog();
```

Polymorphism + Abstract class

```
abstract class Animal {
    protected String name;
    abstract public void say();
}

class Cat extends Animal {
    private void meow() { ... }
    public void say() { meow(); }
}

abstract class Canine extends Animal{
    protected bool likeBones;
}

class Dog extends Canine {
    private void bark() { ... }
    public void say() { bark(); }
}
```

```
Animal  a = null;
Dog  baduki = new Dog();
Cat  nabi = new Cat();

a = baduki; a.say();
a = nabi;  a.say();

Animal c1 = new Animal(); //Compile Error!
Animal c2 = new Cat();
Cat c3 = new Animal(); //Compile Error!
Cat c4 = new Cat();

Animal d1 = new Canine(); //Compile Error!
Animal d2 = new Dog();
Canine d3 = new Dog();
Canine d4 = new Cat(); //Compile Error!
Dog d5 = new Canine(); //Compile Error!
Dog d6 = new Dog();
```

How a decision is made about which method to run

- Step 1: If there is a concrete method for the operation in the current class, run that method.
- Step 2: Otherwise, check in the immediate superclass to see if there is a method there; if so, run it.
- Step 3: Repeat step 2, looking in successively higher superclasses until a concrete method is found and run.
- Step 4: If no method is found, then there is an error
  - In Java and C++ the program would not have compiled

What is an Interface?

- An interface is similar to an abstract class with the following exceptions:
  - All methods defined in an interface are abstract; Interfaces cannot contain any implementation
  - Interfaces cannot contain instance variables.
  - However, they can contain public static final variables (i.e. constant class variables)
- Interfaces are declared using the "interface" keyword
  - If an interface is public, it must be contained in a file which has the same name.
- Interfaces are more abstract than abstract classes
- Interfaces are implemented by classes using the "implements" keyword.

Declaring an Interface

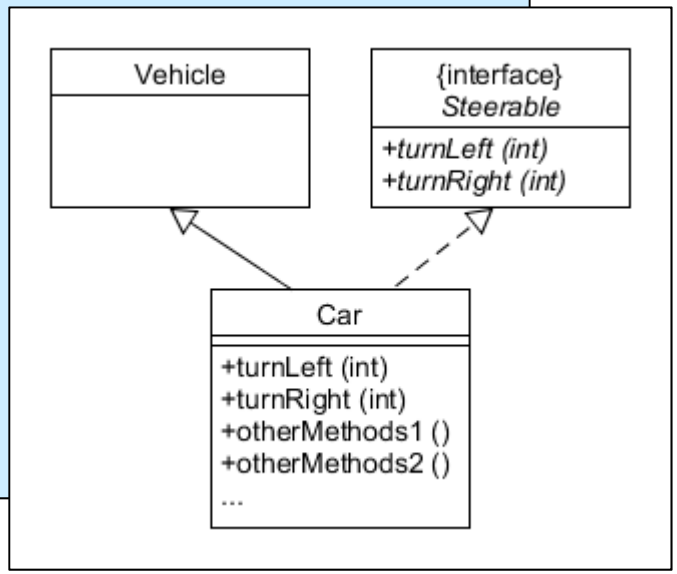
In Steerable.java:

```
public interface Steerable
{
  public void turnLeft(int degrees);
  public void turnRight(int degrees);
}
```

In Car.java:

```
public class Car extends Vehicle implements Steerable
{
  public void turnLeft(int degrees)
  {   …              }
  public void turnRight(int degrees)
  {   ...                }
  public otherMethods1()  { ... }
  public otherMethods2()  { ... }
     …
 }
```

When a class "implements" an interface, the compiler ensures that it provides an implementation for all methods defined within the interface.



Interfaces as Types

- When a class is defined, the compiler views the class as a new type.

- The same thing is true of interfaces. The compiler regards an interface as a type.
  - It can be used to declare variables or method parameters

Abstract Classes Versus Interfaces

- When should one use an Abstract class instead of an interface?
  - If the subclass-superclass relationship is genuinely an "**is a**" relationship.
  - If the abstract class can provide an implementation at the appropriate level of abstraction

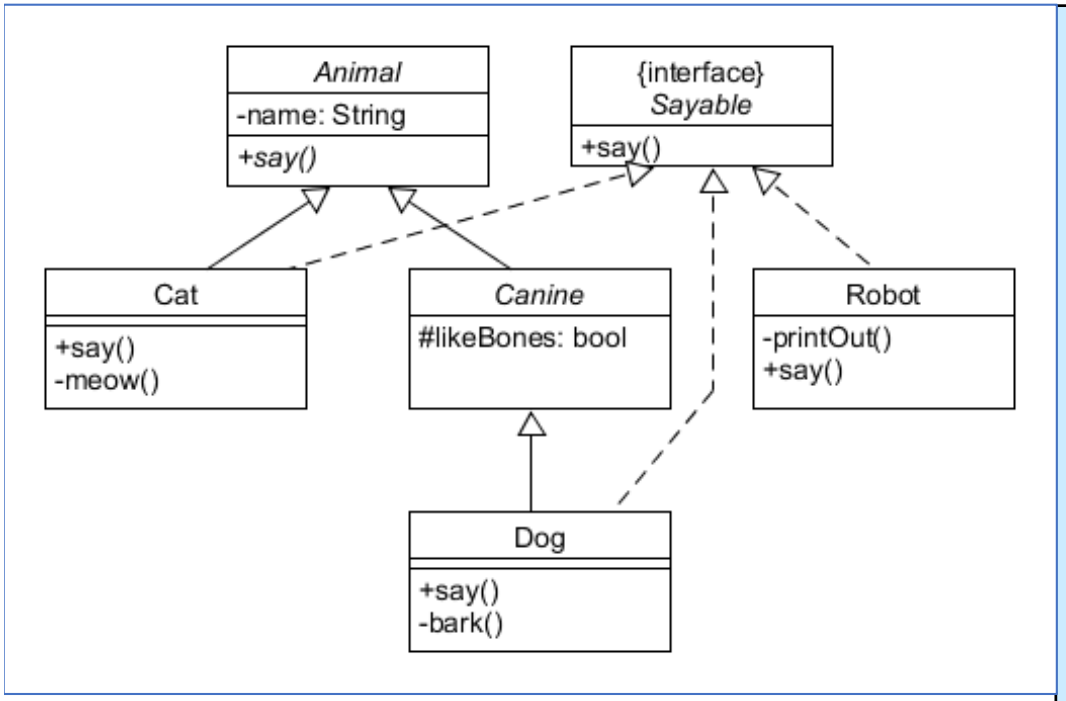
- When should one use an interface in place of an Abstract Class?
  - When the methods defined represent a small portion of a class
  - When the subclass needs to inherit from another class
  - When you cannot reasonably implement any of the methods

Polymorphism with Interface

```
abstract class Animal {
    protected String name;
    abstract public void say();    }

class Cat extends Animal implements Sayable {
    private void meow() { ... }
    public void say() { meow(); }   }

abstract  class Canine extends Animal {
    protected boolean likeBones;  }

class Dog extends Canine implements Sayable {
    private void bark() { ... }
    public void say() { bark(); }   }

class Robot implements Sayable {
    private void printOut() { ... }
    public void say() { printOut(); }  }

interface Sayable {
    public void say();
}
```

```
Dog  baduki = new Dog();
Cat  nabi = new Cat();
Robot robo = new Robot();

Animal  aref = null;
Sayable sref = null;
Canine cref = null;

aref = baduki; aref.say();
aref = nabi; aref.say();
aref = robo; aref.say();

sref = baduki; sref.say();
sref = nabi; sref.say();
sref = robo; sref.say();

cref = baduki; cref.say();
cref = nabi; cref.say();
cref = robo; cref.say();
```

Polymorphism with Interface



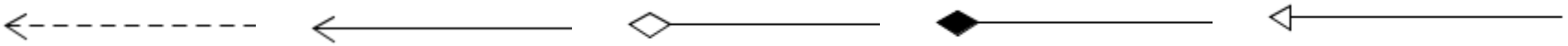

```
Dog  baduki = new Dog();
Cat  nabi = new Cat();
Robot robo = new Robot();

Animal  aref = null;
Sayable sref = null;
Canine cref = null;

aref = baduki; aref.say();
aref = nabi; aref.say();
aref = robo; aref.say();

sref = baduki; sref.say();
sref = nabi; sref.say();
sref = robo; sref.say();

cref = baduki; cref.say();
cref = nabi; cref.say();
cref = robo; cref.say();
```

Polymorphism with Interface

```
abstract class Animal {
    protected String name;
    abstract public void say();    }

class Cat extends Animal implements Sayable {
    private void meow() { ... }
    public void say() { meow(); }   }

abstract  class Canine extends Animal {
    protected boolean likeBones;  }

class Dog extends Canine implements Sayable {
    private void bark() { ... }
    public void say() { bark(); }   }

class Robot implements Sayable {
    private void printOut() { ... }
    public void say() { printOut(); }  }

interface Sayable {
 public void say(); }
```

```
Dog  baduki = new Dog();
Cat  nabi = new Cat();
Robot robo = new Robot();

Animal  aref = null;
Sayable sref = null;
Canine cref = null;

aref = baduki; aref.say();
aref = nabi; aref.say();
aref = robo; aref.say();  // Error!

sref = baduki; sref.say();
sref = nabi; sref.say();
sref = robo; sref.say();

cref = baduki; cref.say();
cref = nabi; cref.say(); // Error!
cref = robo; cref.say(); // Error!
```

Class Relationships

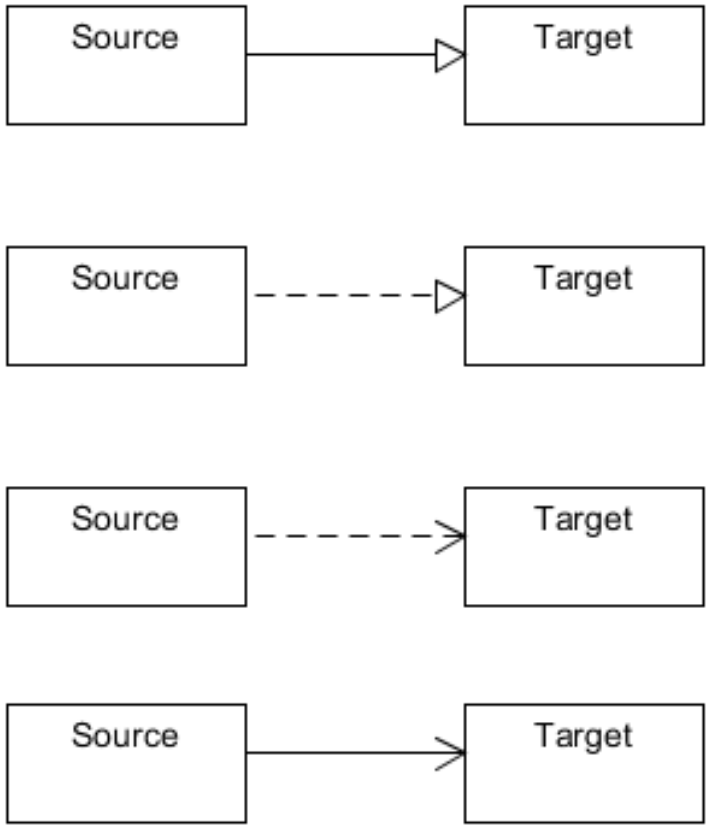


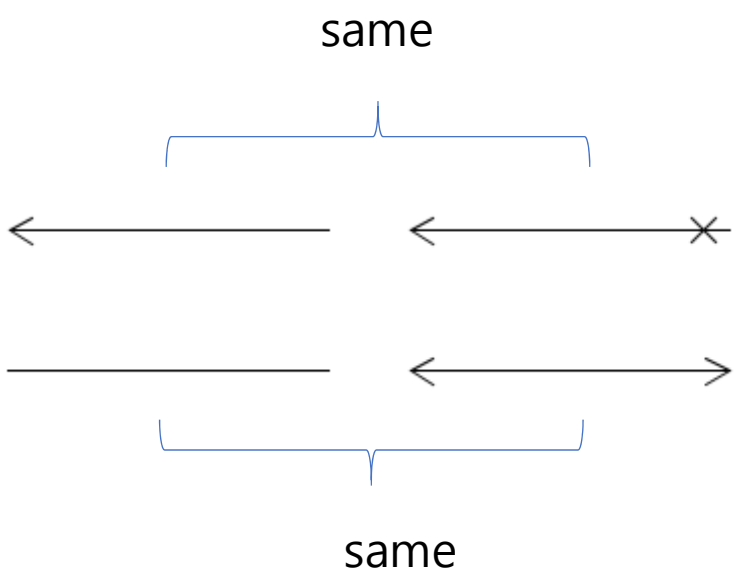
Dependency	Association	Aggregation	Composition	Inheritance
dashed arrow	simple connecting line	empty diamond arrow	filled diamond arrow	empty arrow
when objects of one class work briefly with objects of another class	when objects of one class work with objects of another class for some prolonged amount of time	when one class owns but shares a reference to objects of another class	when one class contains objects of another class	when one class is a type of another class

weaker relationship

stronger relationship

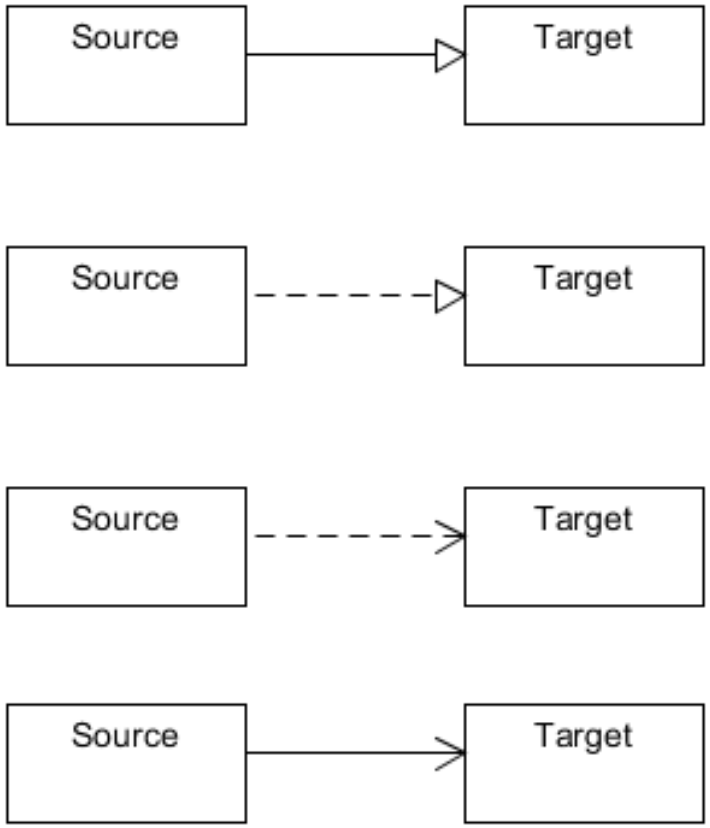
Class Relationships

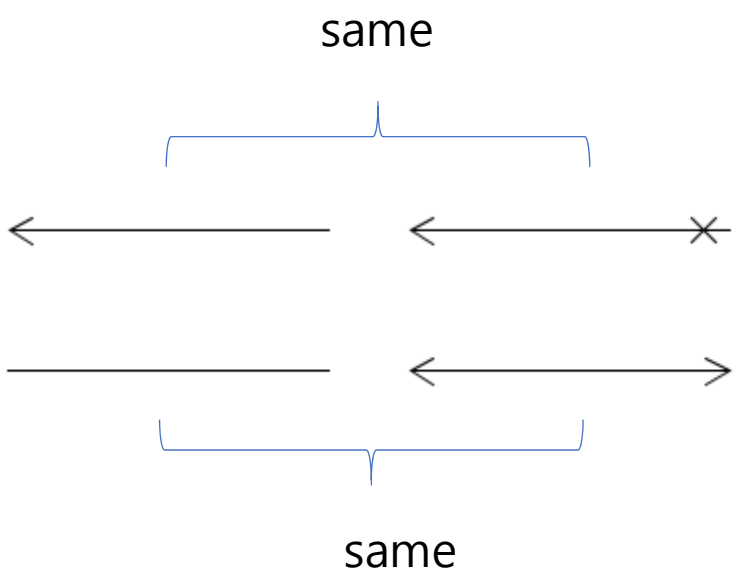




If Target changes, then Source may be affected.
Target is ignorant of Source.

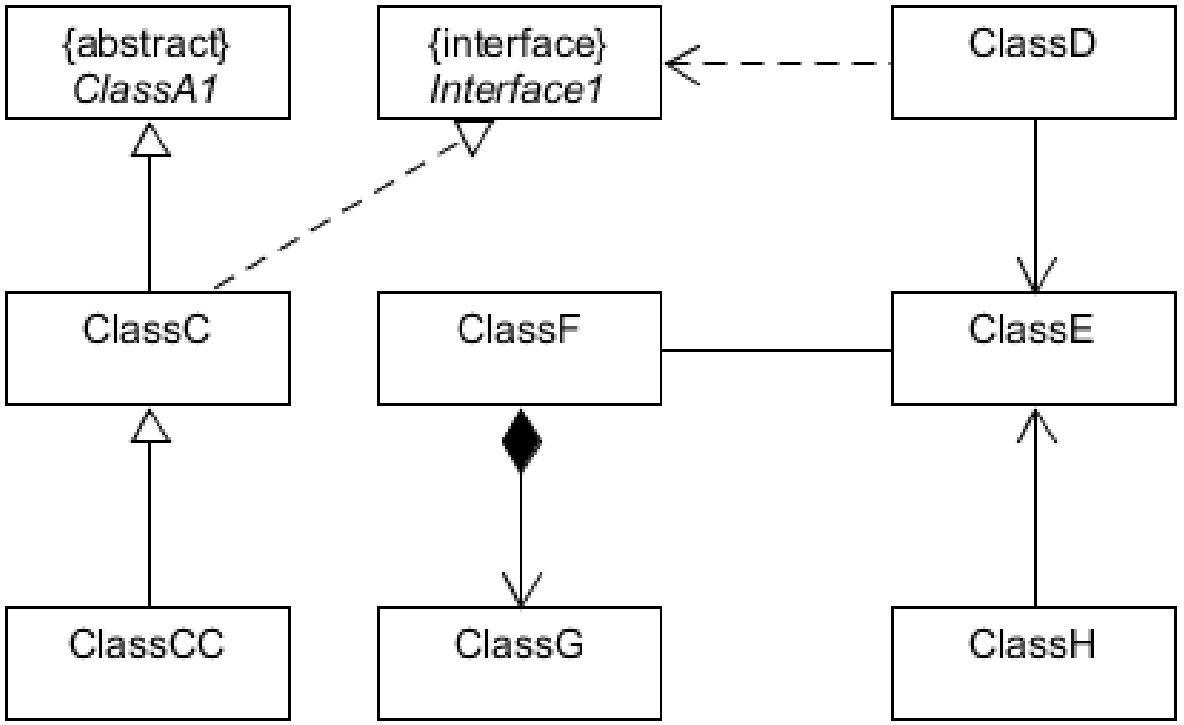
Class Relationships and Change Propagations





If Target changes, then Source may be affected.
Target is ignorant of Source, hence the change of Source is not propagated.

Class Change Propagations



(start) ClassA1 -> ClassC -> ClassCC (end)
(start) ClassD (end)
(start) Interface1 -> ClassC, ClassD -> ClassCC
(start) ClassG -> ClassF -> ClassE -> ClassD, ClassH
(start) ClassE -> ClassD, ClassH, ClassF (end)

Encapsulation

- Design principles for hiding internal design decisions related to the selected algorithm and data structures of a module from the outside world.
  - the internal design decisions are most likely to change
  - Thus, reduce the side effects of any future maintenance or modification of the design and hence minimizing the effect on the other modules in the design.
- Information hiding : Encapsulation = Principle : Technique

Summary

- Object-Oriented Paradigm
  - class, inheritance, polymorphism
- Polymorphism
  - method overloading
  - method overriding
- Class Relationships and Change propagations