LESSON VII.
Overriding, Abstract class and Interface

Trinh Thanh TRUNG (MSc)
trungtt@soict.hust.edu.vn
094.666.8608

Objectives

- Understand and master some Java techniques for realizing the inheritance

Content

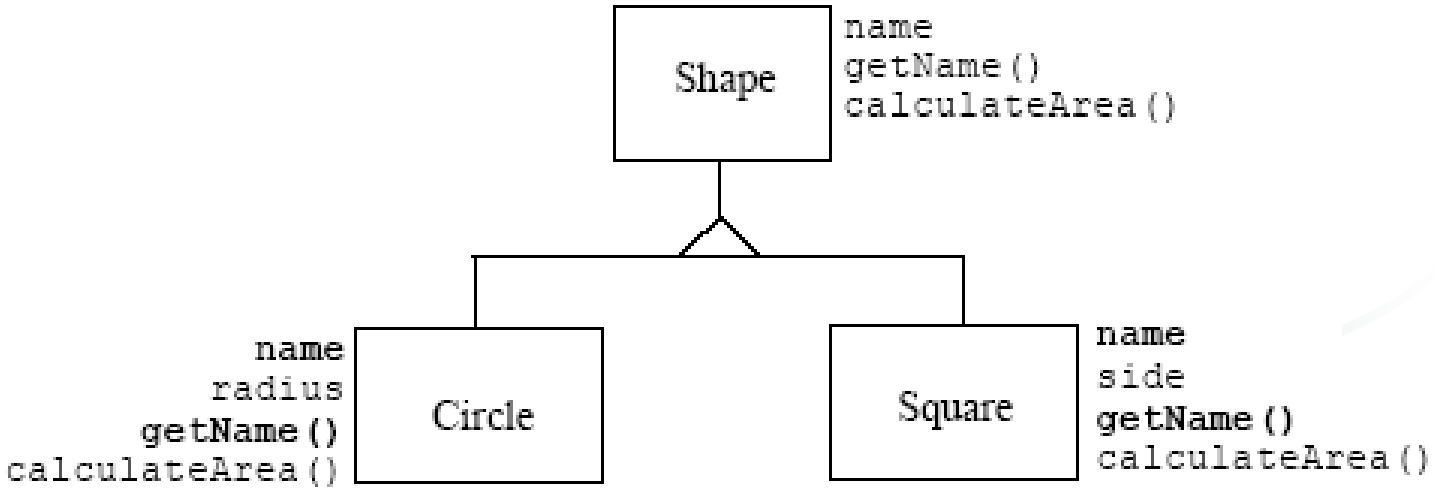
- Method overriding
- Single inheritance and multiple inheritance
- Abstract class and abstract method
- Interface and implementation

I. METHOD OVERRIDING

1. Concept
2. Final modifier and overriding
3. Object class

1. Concept

- The sub class redefine a method that is inherited from a super class
- The redefined method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is called.



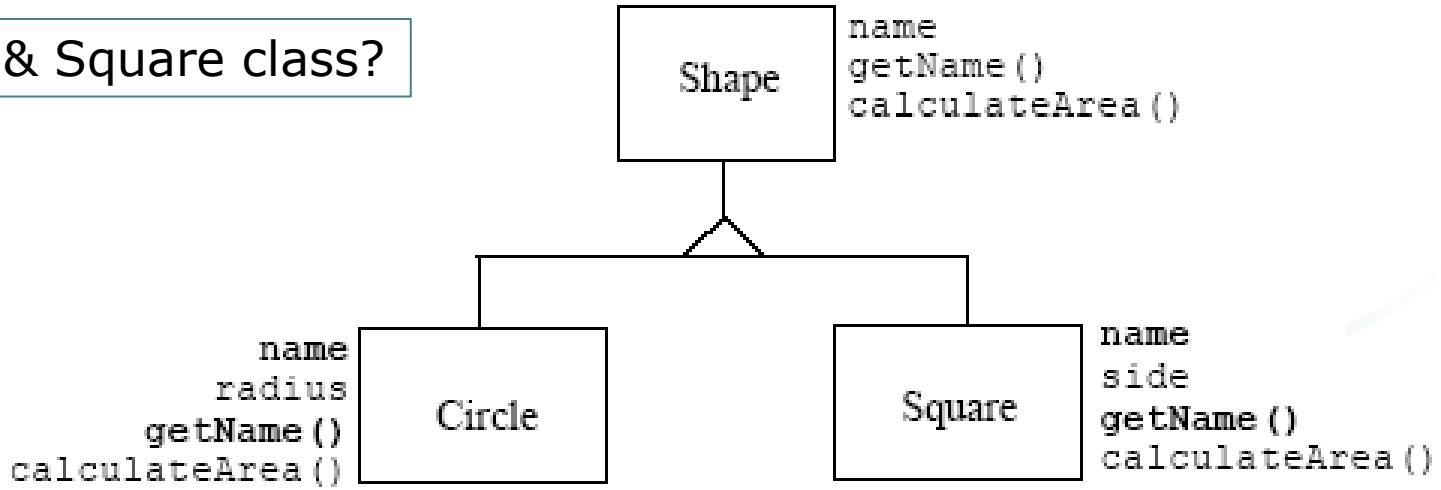
Method overriding: example

```
class Shape {
    protected String name;
    Shape(String n) { name = n; }
    public String getName() { return
name; }
    public float calculateArea() {
return 0.0f; }
}
```

```
class Circle extends Shape {
    private int radius;
    Circle(String n, int r){
        super(n);
        radius = r;
    }

    public float calculateArea() {
        float area = (float) (3.14 * radius *
radius);
        return area;
    }
}
```

Triangle & Square class?



Method overriding: example

```
// Account class
class Account {
     // Member variables
     protected String owner;
     protected long balance;

     //…

     public void display() {
     System.out.print("Owner:" +
     owner);
     System.out.println("\t Balance:"
     + balance);
     }
}
```

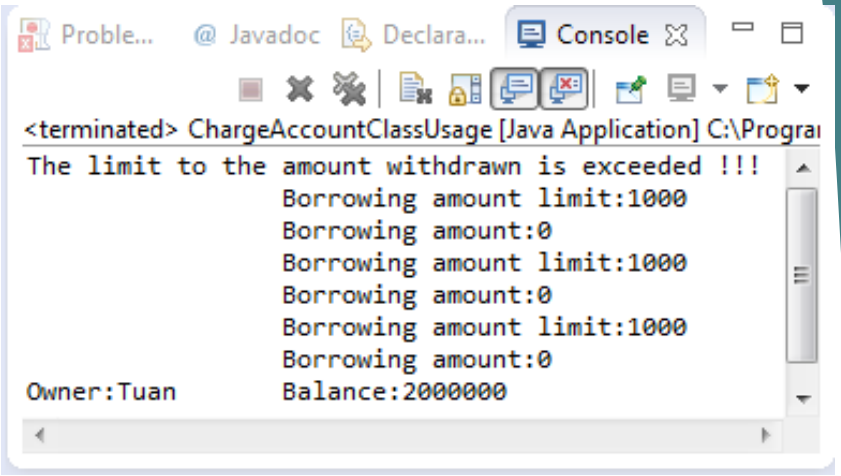
```
public class ChargeAccount extends Account{
     // Additional member variables
          private int overdraft;
          private int overdraft_limit;
          //…
          // access to the super class' member


     public void display() {
          System.out.println("\t\t Borrowing
          amount limit:"+ overdraft_limit);
          System.out.println("\t\t Borrowing
          amount:" + overdraft);
     }
}
```

Method overriding: example

```
public class ChargeAccountClassUsage {
     public static void main(String[] args) {
          // creat a super class object
          Account account = new Account();
          // create and initiate a sub class object
          ChargeAccount chargeacc =
                    new ChargeAccount();
          chargeacc.setData("Giang", 1000000);
          chargeacc.setOverdraftLimit(1000);
          chargeacc.loan(2000);
          // (1) call overridden method
          chargeacc.display();
          // (2) can not call method from its super class, once it is overridden
          // why ? The object executing the method is of type ChargeAccount
          ((Account) chargeacc).display();
          Account account1;
          account1 = chargeacc;
          account1.display(); // (3) using display() of ChargeAccount
          account1 = account;
          account1.display(); // (4) using display() of Account

     }
}
```



2. Final modifier and overriding

- A class may be declared as final
  - that class may not be extended
- A method in a class may be declared as final
  - that method may not be overridden
  - guarantees behavior in all descendants
  - can speed up a program by allowing static binding (binding or determination at compile time what code will actually be executed)
- All static methods and private methods are implicitly final, as are all methods of a final class.

Final modifier in inheritance: example

```
// Account class
final class Account {
     // Member variables
     protected String owner;
     protected long balance;

     // Declaring a class final
     // implicitly declares
     //  all its methods final
     public void display() {
     System.out.print("Owner:" +
     owner);
     System.out.println("\t Balance:"
     + balance);
     }
}

// It is illegal to declare a class
//  as both abstract and final.
final abstract class Account {
...
}
```

```
public class ChargeAccount extends Account{
     // Additional member variables
          private int overdraft;
          private int overdraft_limit;
          //…
          // access to the super class' member


     public void display() {
          System.out.println("\t\t Borrowing
          amount limit:"+ overdraft_limit);
          System.out.println("\t\t Borrowing
          amount:" + overdraft);
     }
}
```

Overriding vs. Overloading

- Same signature
  - Identical method name
  - Identical parameter lists
  - Identical return type
- Defined in two or more classes related through the inheritance
- In a class: one overriding method per overridden method

- Different signatures
  - Identical method name
  - Different parameter lists (types, number)
  - Identical/ different return type (can not overload on return type)
- Defined in the same class
- In a class: several overloading methods per overloaded method

3. Object Class

- Object class is a super-class of all Java classes:
  - Object is the root of the Java inheritance hierarchy.
  - A variable of the Object type may refer to objects of any class.
  - As arrays are implemented as objects, it may also refer to any array.

Overriding the Object class' methods

Methods that can be overridden

- `Object clone()`
- `void finalize()`
- `int hashCode()`
- **`String toString()`**
- **`boolean equals(Object object)`**

Methods that can not be overridden

- `void notify()`
- `void notifyAll()`
- `Class getClass()`
- `void wait()`
- `void wait(long milliseconds)`
- `void wait(long milliseconds, int nanoseconds)`

Overriding rules

- The overriding method in sub class MUST
  - Has the same parameter list
  - Has the same return type
- The overrided method in super class MUST NOT
  - Be **final** or **static**
  - Has **private** modifier
- The access modifiers for overriding methods in sub class MUST NOT be **stricter than ones in parent class**
  - E.g. if the method in parrent class is **protected,** the overriding method in sub class must NOT be **private**

Method overriding: example

```
class Parent {
   public void doSomething() {}
   protected int doSomething2() {
        return 0;
   }
}
class Child extends Parent {
   protected void doSomething() {}
   protected void doSomething2() {}
}
```

Unable to override: Not the same return type

Unable to override: Stricter access modifier

II. ABSTRACT CLASSES AND ABSTRACT METHODS

1. Abstract class
2. Abstract method
3. Example

1. Abstract class

- An abstract class provides an outline from which other classes inherit attributes and operations
  - Provide implementation for some of methods that it declares
  - Subclasses that extend it must complete the class definition

  → Can not create instances of abstract classes
- Syntax

```
public abstract class ClassName{
   // definition of concrete methods
   // declaration of abstract methods
}
```

2. Abstract method

- Abstract methods are methods that do not have implementation (body)

```
public abstract
    return-type method-signature;
```

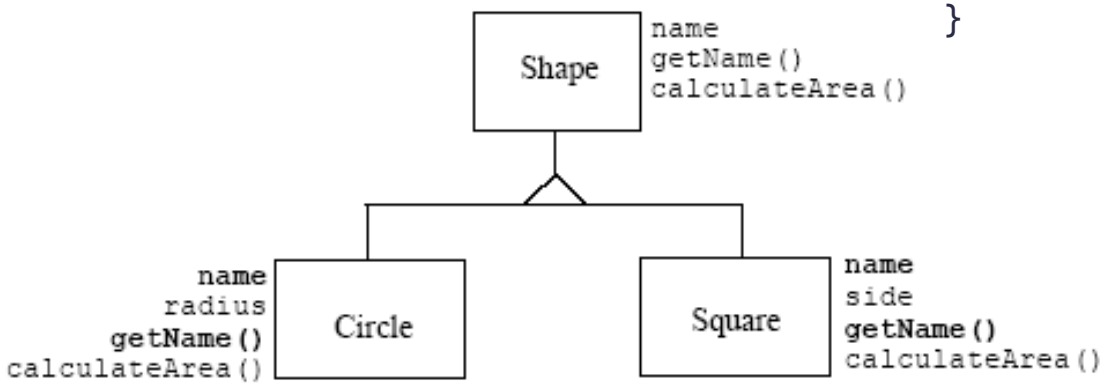
→ To become concrete class, a sub class of an abstract class must implement all abstract methods of super abstract classes in the inheritance chain.

→ Otherwise, this sub class will become an abstract class and can not be instantiated.

3. Example

```
abstract class Shape {
    protected String name;
    Shape(String n) { name = n; }
    public String getName() { return name; }
    public abstract float calculateArea();
}
```

```
class Circle extends Shape {
    private int radius;
    Circle(String n, int r){
      super(n);
      radius = r;
    }
    public float calculateArea() {
     float area = (float) (3.14 * radius *
    radius);
     return area;
    }
}
```

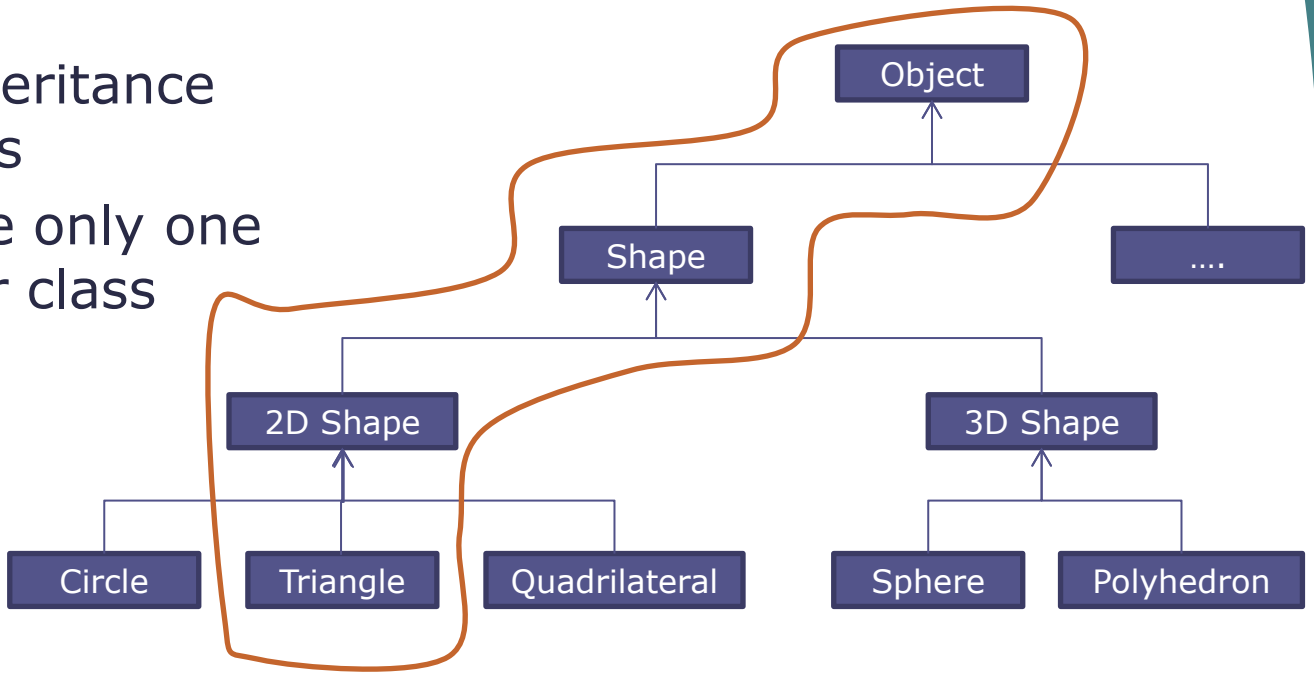


III. SINGLE INHERITANCE AND MULTIPLE INHERITANCE

1. Inheritance chain
2. Single and multi-level inheritance
3. Multiple inheritance

Inheritance chain

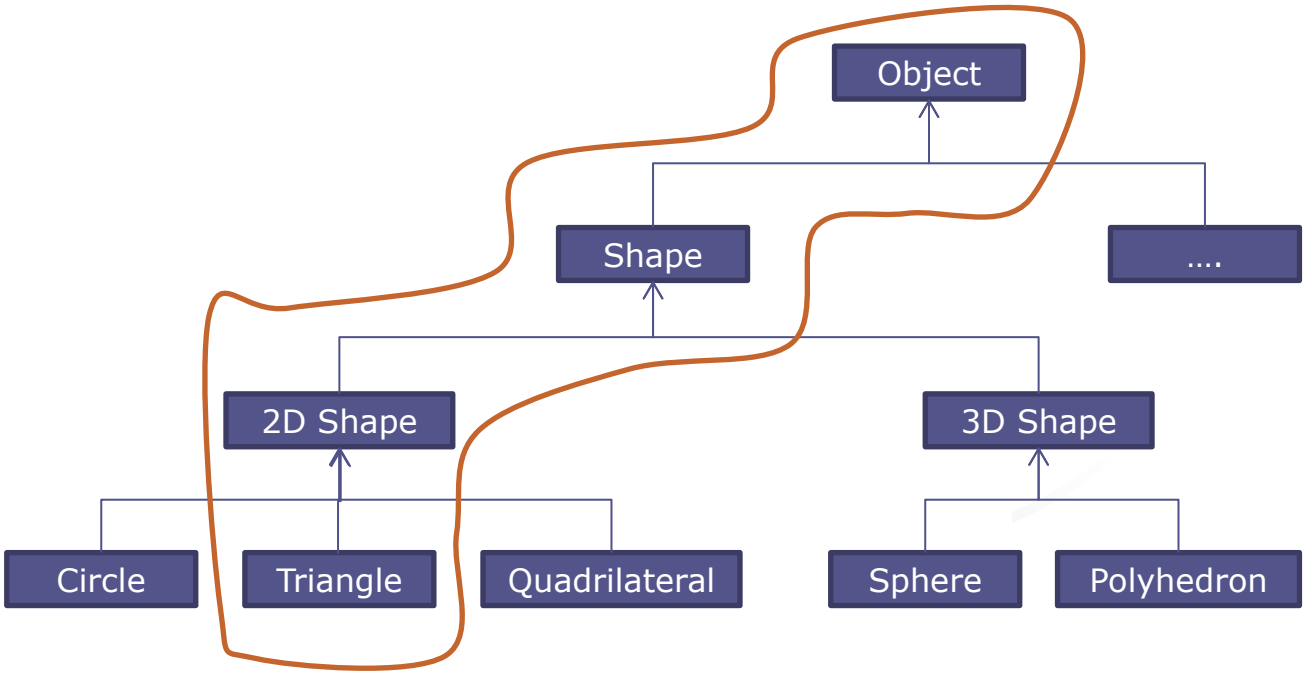
- The path of inheritance over the classes
- Each class have only one parent or super class



Single and multi-level inheritance

- Single inheritance:
  - there are only one direct super class from which a subclass explicitly inherits.
- Example :
  - Triangle and 2DShape
  - Shape and Object

- Multi-level inheritance:
  - A subclass inherits from any class above its direct super class in the class hierarchy
- Example:
  - Triangle and Object
  - Triangle and Shape



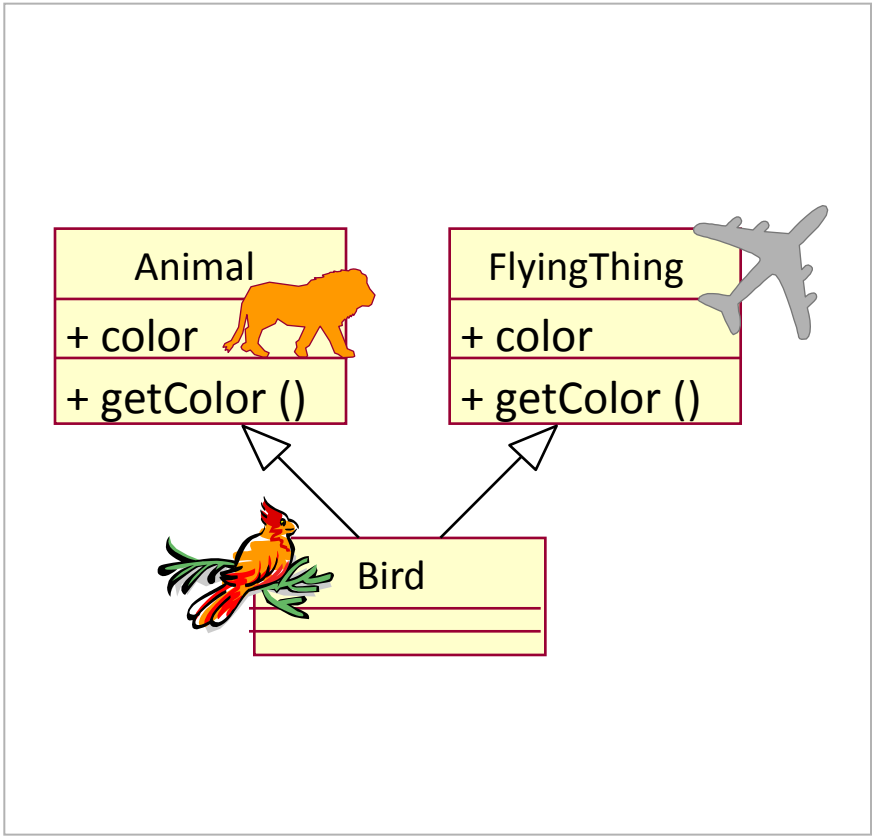
Multiple inheritance

- A sub class in the hierarchy inherits from more than one super classes in more than one inheritance path.

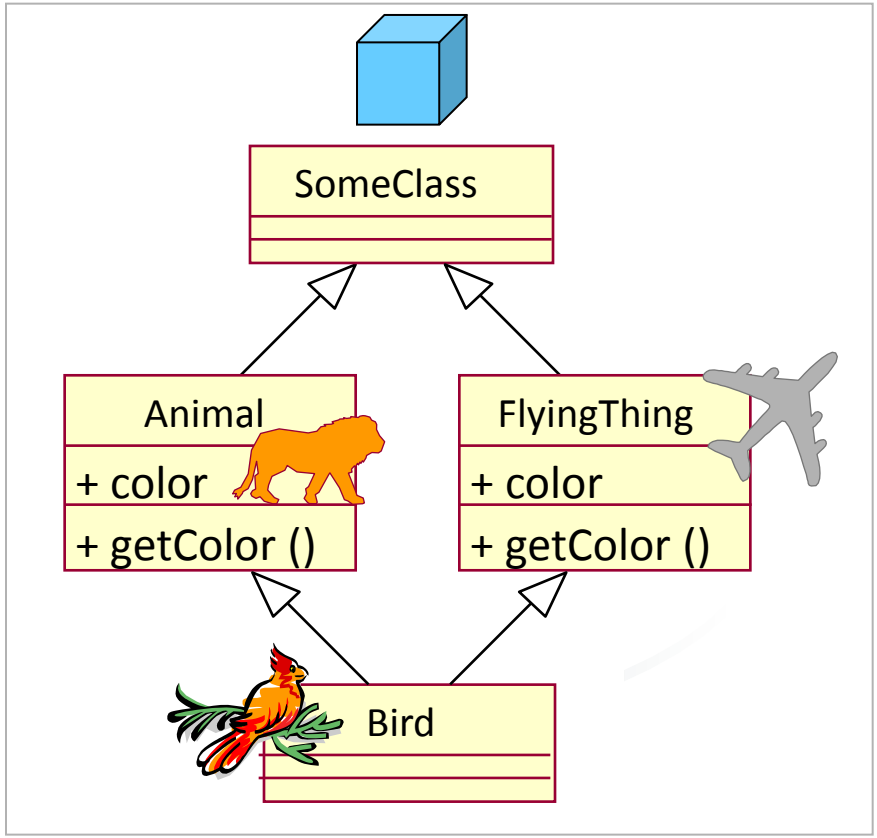
Multiple inheritance

- Problem with multiple inheritance

Name clashes on attributes or operations



Repeated inheritance



IV. INTERFACE AND IMPLEMENTATION

1. Interface

1. Interface

- An interface defines a standard and public way of specifying the behavior of classes
  - Classes that implement an interface must respect the methods' return type and the signature as declared.
  - All declared method must be implemented
- Syntax

```
[access-modifier] interface interface-name {
   // variables (constant data)
   // implicitly abstract and public methods
}
```

Interface Variables declaration

- Syntax

```
public static final type-name var-name = constant-expr;
```

- This is a technique to import shared constants into multiple classes:
  - declare an interface with variables initialized to the desired values
  - include that interface in a class through implementation
- Variables declared in an interface must be constants
  - Every variable declaration in the body of an interface is implicitly public, static, and final
- If the interface does not declare any method, the class does not implement anything except importing the variables as constants.
- Example :

```
public interface TwoDimensionShape {
   int DIMENSION = 2;
}
```

Interface methods declaration

- **`[access-modifier] return-type method-name (parameters list);`**
- Methods declared in an interface are implicitly abstract an public
  - public: all others can be accessed
  - abstract: no implementation

→ An interface can be considered as an abstract class which contains only abstract methods.

Interface Inheritance

- One interface may inherit another interface.
- The method and the constant that are defined in the super interface are inherited to the sub-interface.
- The inheritance syntax is the same for classes and interfaces.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

```
public interface Shape {
   public String getName();
   public void display();
}
public interface TwoDimensionShape
   extends Shape{

   public double calculateArea();

   public double
   calculatePerimeter();
}
```

Example

```
public interface Shape {
   public String getName();
   public void display();
}

public interface
   TwoDimensionShape extends
   Shape{
        public double
        calculateArea();
        public double
        calculatePerimeter();
}
```

```
public class Triangle implements
   TwoDimensionShape{
   // declare variable here
   private String name;
   /*   all of methods of Shape
        and TwoDimensionShape
        must be
        implemented here
        in a specific  way. */
…
}
```

Why interface?

- To model multiple inheritance
- To reveal an object's programming interface (functionality of the object) without revealing its implementation
- To have unrelated classes implement similar methods (behaviors)

Abstract class vs. interface

- Mix of abstract and concrete methods
  - Abstract methods must be declared explicitly using abstract keyword
- Contain attributes that are inherent to a class
- Have one direct inherited relationship with its super class

- All methods are abstract methods
  - a subclass is required to implement them
- Can only define constants
- Interfaces have no direct inherited relationship with any particular class, they are defined independently
  - Interfaces themselves have inheritance relationship among themselves

2. Interface implementation

- Syntax:

```
[modifier] class class_name
   [extends super-class-name]
   implements   comma-separated-list-of-interfaces {
        …
        //      overridden methods
        //      its own methods
        …
}
```

- A concrete class can only extend one super class, but it can implement multiple interfaces
  - It is required to implement all abstract methods of all interfaces

Example: Using interface as a type

```
public interface Shape {
    public String getName();
    public void display();
}

public interface TwoDimensionShape
    extends Shape{
    public double calculateArea();
    public double calculatePerimeter();
}

public class Triangle implements
    TwoDimensionShape{
    …
}
```

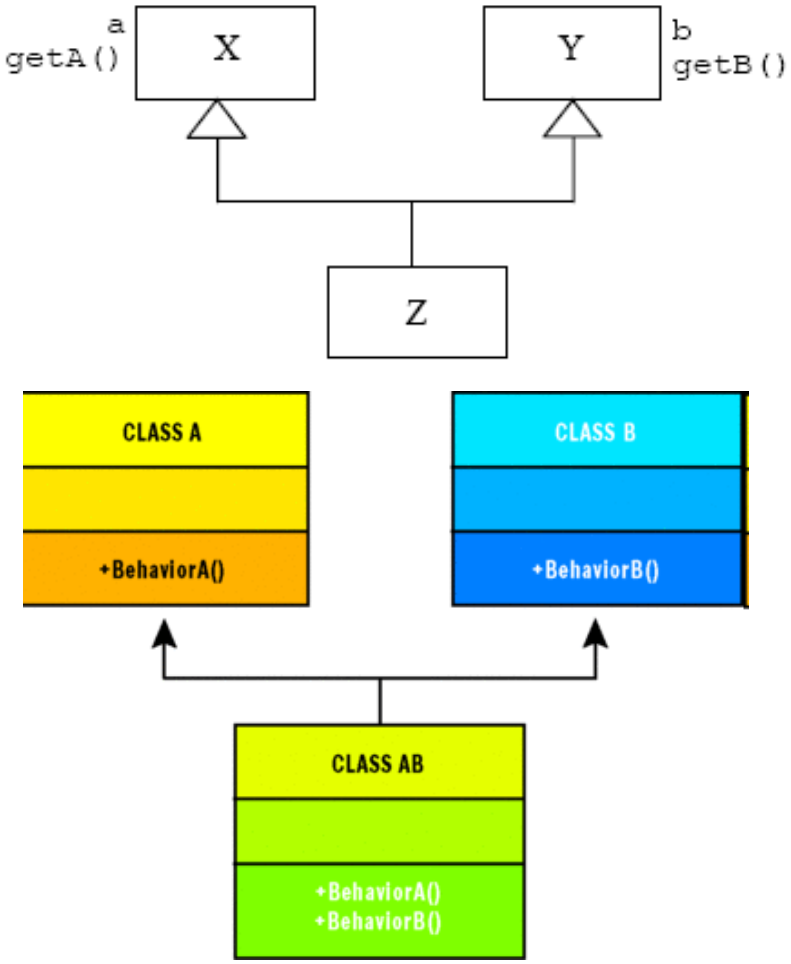
```
public class TriangleUsage {
     ...
    // legal to create a variable with
    // the interface type
    TwoDimensionShape tsh;

    // legal to refer to any object of any
    // class implementing this interface
    Shape tri = new Triangle();
    TwoDimensionShape tsh1 = new Triangle();

    // call any method in the interface
    // using the interface type variable
    tri.display();
    double area = tsh1.calculateArea();
}
```

Disadvantages

- Does not provide a natural solution for non-conflicted cases
- Unable to reuse the code



Quiz

Review

- Method overriding:
  - The sub class redefine a method that is inherited from a super class.
- Single inheritance and multiple inheritance
  - Single inheritance + multi-level inheritance: create a new class as an extension of another class using extends keyword
    - Class could be either concrete or abstract
  - Multiple inheritance: create a new class to implement the methods that are defined as part of an interface using implements keyword
    - Class could implement more than one interface

Review

- Abstract class and abstract method
  - Abstract class: outline from which other classes inherit attributes and operations.
  - Abstract method: no implementation
- Interface and implementation
  - Interface: what a class must do
  - Interface implementation: complete set of methods defined by this interface