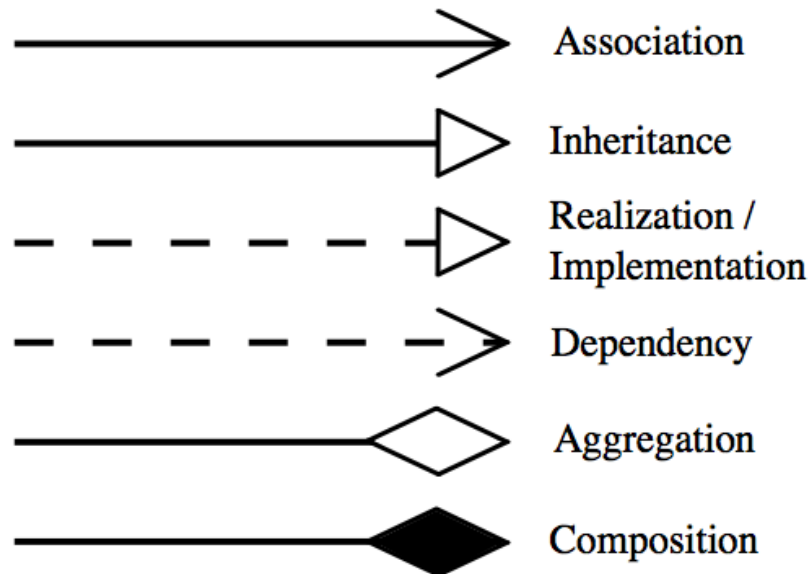


04. OO Relationships





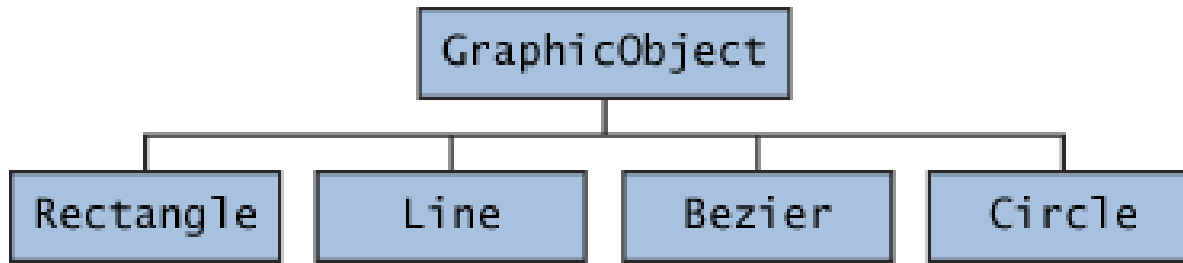
Summary: Inheritance

- Except for the Object class, a class has exactly one direct superclass. A class inherits fields and methods from all its superclasses, whether direct or indirect.
- An abstract class can only be subclassed; it cannot be instantiated. An abstract class can contain abstract methods—methods that are declared but not implemented. Subclasses then provide the implementations for the abstract methods.



Summary: Inheritance

- You can prevent a class from being subclassed by using the final keyword in the class's declaration. Similarly, you can prevent a method from being overridden by subclasses by declaring it as a final method.



```
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int
        newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}
```

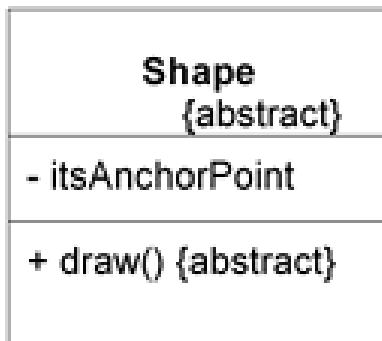
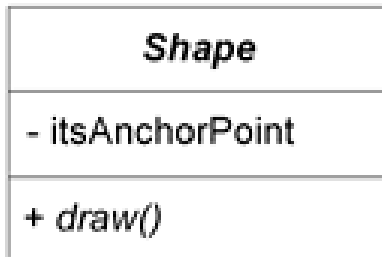
```
class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

```
class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```



abstract class

In UML there are two ways to denote that a class or a method is abstract. You can write the name in italics, or you can use the {abstract} property.



```
public abstract class Shape
{
    private Point itsAnchorPoint;
    public abstract void draw();
}
```



Realization/Implementation

- Realization is a relationship between two elements in a UML diagram where one element specifies behavior and the other element implements or executes, in other words, realizes, that behavior.
- There is a source element, called the realization element, and a target element, called the specification element, and the relationship is also often referred to as being between a supplier and client.



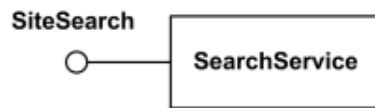
Realization/Implementation

- In many cases, the specification element will be an interface, or a collection of operations, with the realization element as the implementation of those behaviors or operations.

Realization/Implementation

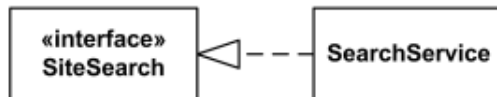
Notation

The **interface realization** dependency from a classifier to an interface is shown by representing the interface by a circle or ball, labeled with the name of the interface and attached by a solid line to the classifier that realizes this interface.



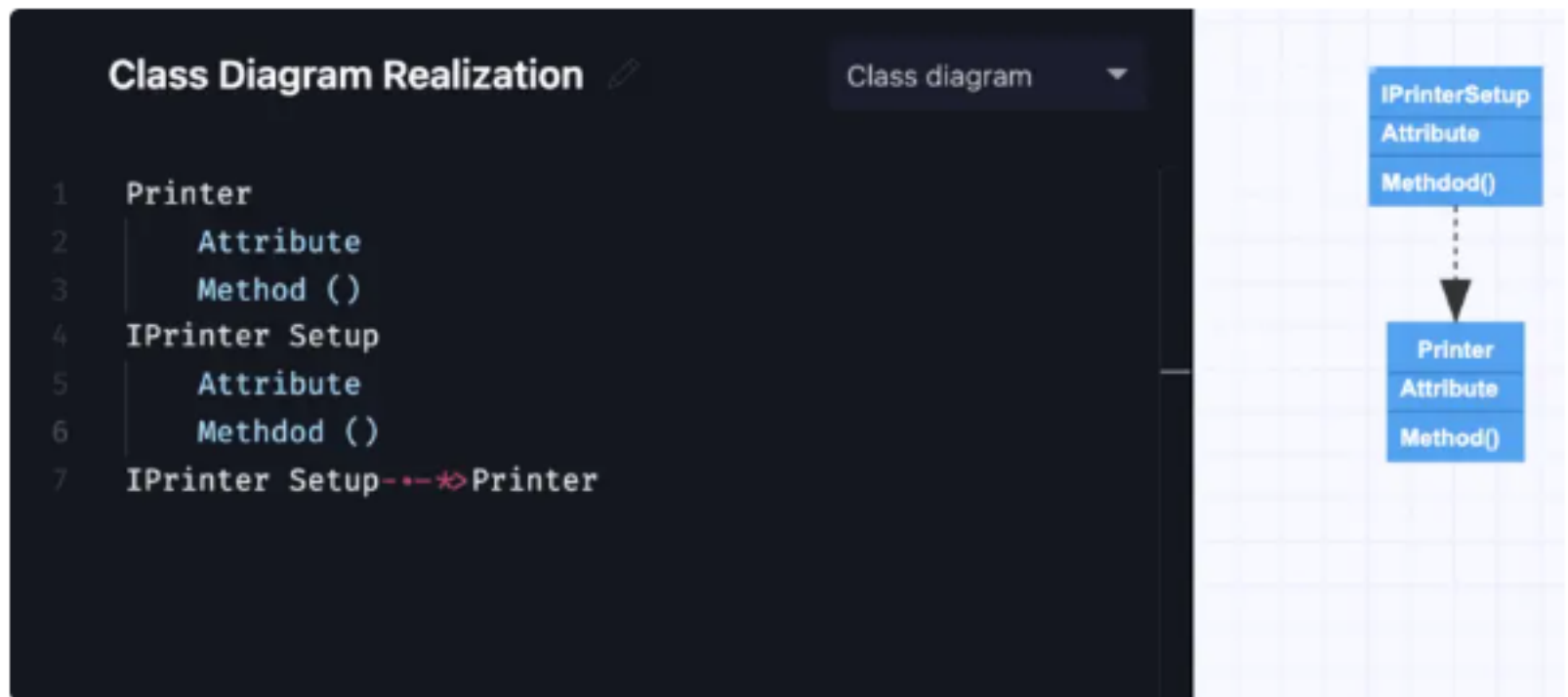
Interface SiteSearch is realized (implemented) by SearchService.

If interface is represented using the rectangle notation, **interface realization** dependency is denoted with interface realization arrow. The classifier at the tail of the arrow implements the interface at the head of the arrow.

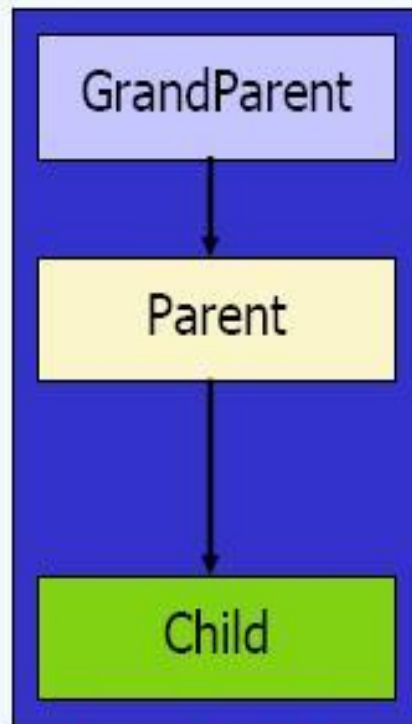


Interface SiteSearch is realized (implemented) by SearchService.

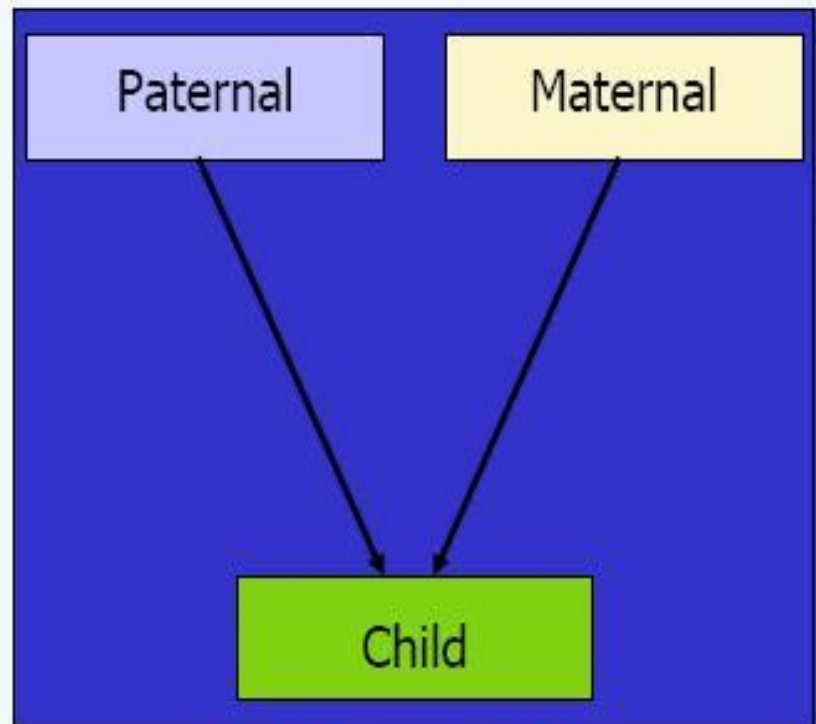
Realization/Implementation



Interfaces in Java



Allowed in Java



Not Allowed in Java





Interfaces in Java

- An ***interface*** is a set of predefined methods to be implemented by one or more classes in future.
- An Interface will just give what the method should do, but it will not give the implementation for it.
- This helps the programmer to write his own logic in his class, which implements the particular interface.
- The methods in an interface will have no body and it just mention the method signature.



Interface

- A completely abstract class
- Only constants and abstract methods are allowed
- The '*class*' keyword is replaced by '*interface*'

```
public interface Drawable {  
    void draw();  
    double getArea();  
}
```



Implementing an Interface

- The general form of implementing an interface is:

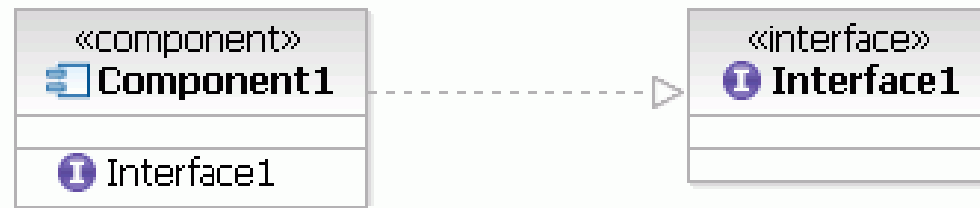
```
class classname [extends superclass] implements  
    interface1 [, interface2, . . .] {  
    // body.....  
}
```



Interface - cont'd

- All members are implicitly public; no need to supply access modifiers (and supplying any other than public is an error).
- Cannot be instantiated.
- Other classes may *'implement'* an interface.

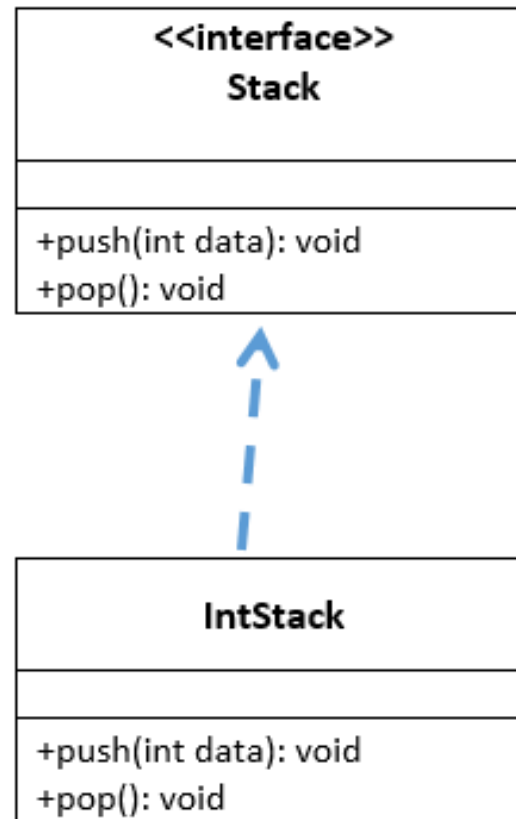
Interface



- An interface realization relationship is displayed in the diagram editor as a dashed line with a hollow arrowhead. The interface realization points from the classifier to the provided interface.



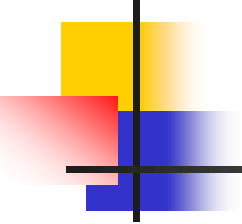
Exercise





Example:

```
public class Circle extends Shape implements Drawable {  
    public void draw() { ... }  
    public double getArea() {  
        return radius*radius*Math.PI;  
    }  
    public display() { . . . }  
}
```



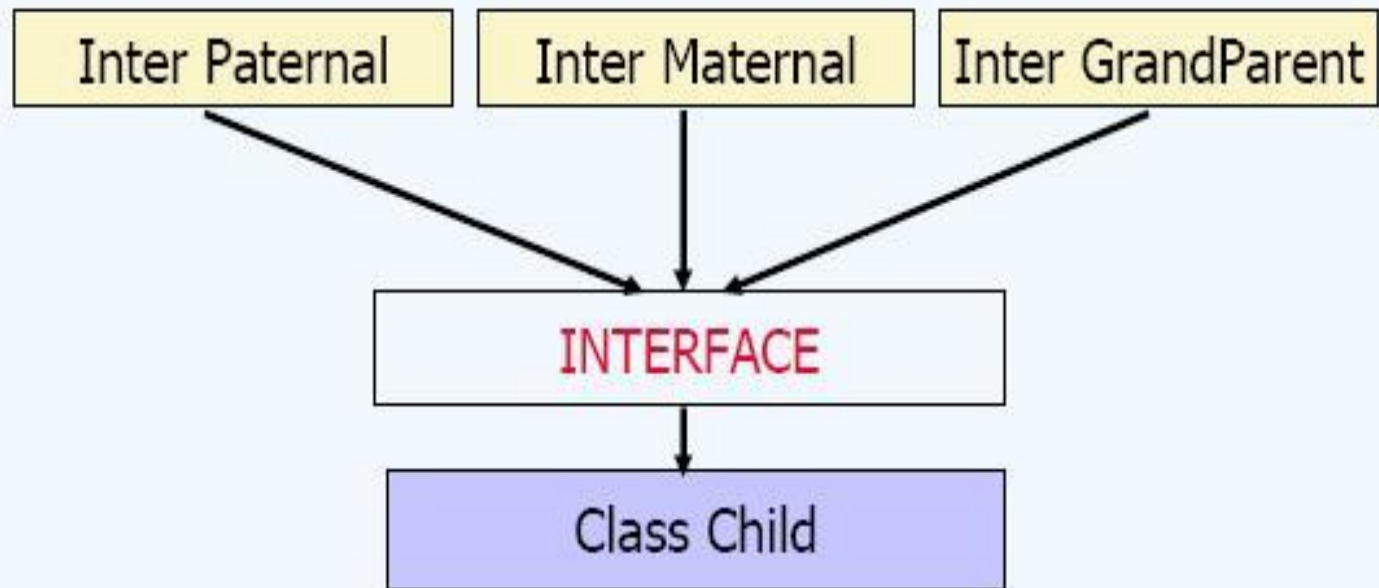
```
abstract class GraphicObject {  
    int x, y;  
    ...  
    void moveTo(int newX, int  
        newY) {  
        ...  
    }  
    abstract void draw();  
    abstract void resize();  
}
```

```
interface GraphicObject {  
  
    void draw();  
    void resize();  
}
```

```
class Circle extends GraphicObject {  
    void draw() {  
        ...  
    }  
    void resize() {  
        ...  
    }  
}
```

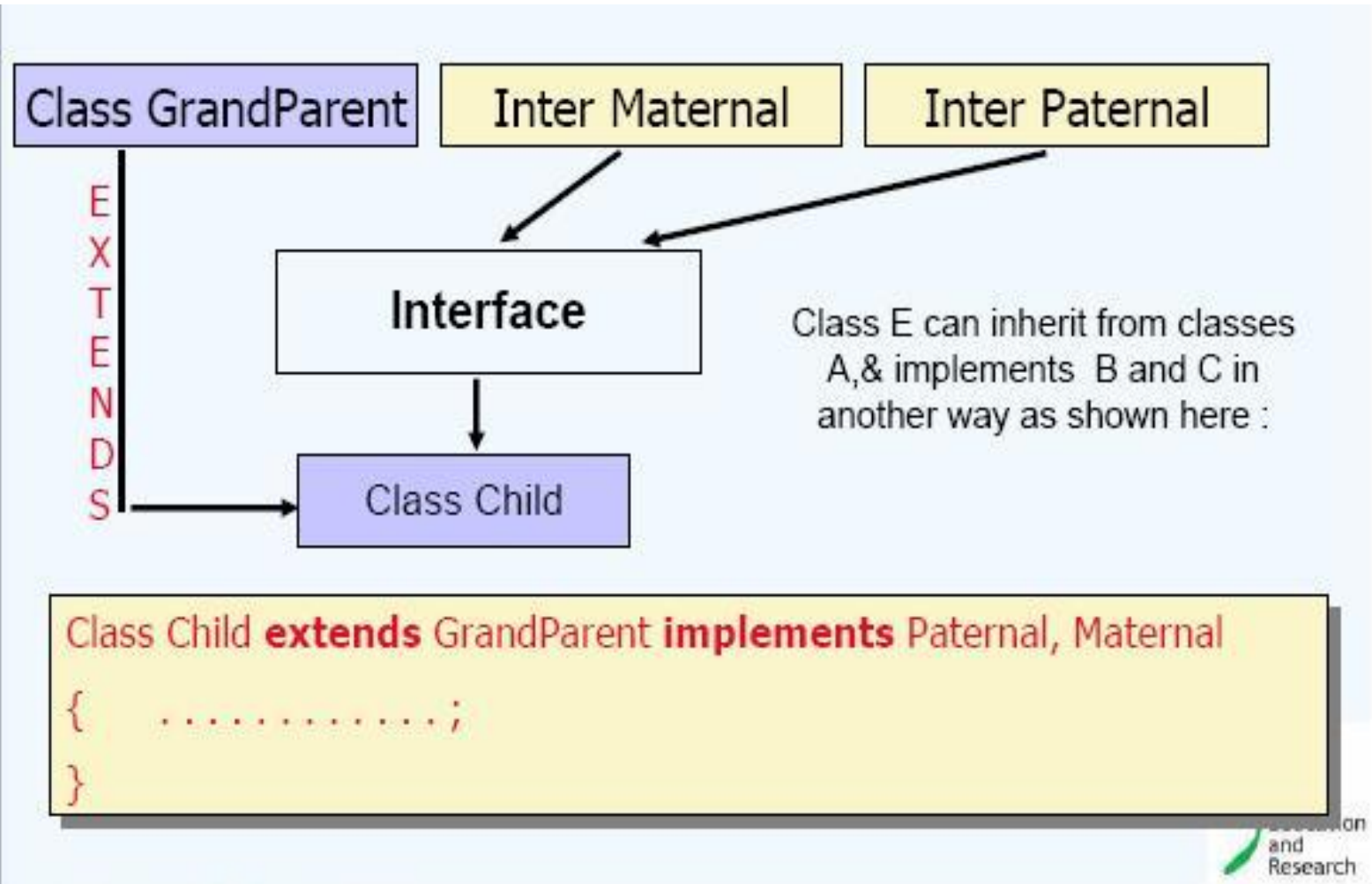
```
class Circle implements GraphicObject {  
    void draw() {  
        ...  
    }  
    void resize() {  
        ...  
    }  
}
```

Implementing Multiple Inheritance in Java



```
Class Child implements Paternal, Maternal, GrandParent  
{ .....;  
}
```

Implementing Multiple Inheritance in Java





Abstract Class v/s Interfaces

- Specifies the full set of methods for an object.
- Implements none, some or all of its methods.
- Useless without being subclassed.

- Specifies a subset of methods for an object.
- Implements none of its methods.
- Useless without being implemented.

Both Abstract classes and Interfaces cannot be instantiated.



Summary:

Classes, Abstract Classes, and Interfaces

- **Classes**

- All declared methods must be defined.
- No restriction on member variables.

- **Abstract Classes**

- Some methods may be defined.
- No restriction on member variables.

- **Interfaces**

- Methods are only declared.
- Member variables must be static and final.



Polymorphism Example

```
public class Animal {  
    public void sound() { }  
}  
  
public class Dog extends Animal {  
    public void sound() {  
        System.out.println("Woof !!");  
    }  
}  
  
public class Duck extends Animal {  
    public void sound() {  
        System.out.println("Quack!!");  
    }  
}
```



Polymorphism Example cont'd

```
public class TryPolymorphism {  
    public static void main(String args[]) {  
        // theAnimals is an array of superclass references  
        Animal[] theAnimals= { new Dog(), new Duck() };  
  
        Animal petChoice = theAnimals[0];  
        petChoice.sound();           // calls Dog's method  
  
        petChoice = theAnimals[1];  
        petChoice.sound();           // calls Duck's method  
    }  
}
```




Polymorphism via Interfaces

- An object of a class implementing an interface may be treated as an object of type corresponding to the interface.
- Different classes support the same set of operations by implementing the same interface.
- At runtime, any of the implementing classes can be used



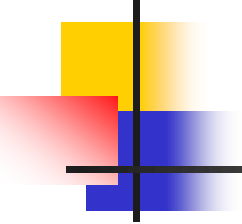
Example:

```
public interface Shape {  
    public double getArea();  
}
```

```
public class Circle implements Shape {  
    public double getArea() { return PI * r * r ; } }
```

```
public class Rectangle implements Shape {  
    public double getArea() { return height * width ; } }
```

```
public class Triangle implements Shape {  
    public double getArea() { return 0.5 * base * height ; } }
```



```
public class ShapeAreaFinder {  
    public static void main(String[] args) {  
        Shape [] theShapes = { new Circle(1, 2, 3), new  
                                Rectangle(10, 23), new Triangle(12, 20) };  
  
        System.out.println(theShapes[0].getArea());  
        System.out.println(theShapes[1].getArea());  
        System.out.println(theShapes[2].getArea());  
    }  
}
```