

# Object-Oriented Programming with Java **Interfaces**



Quan Thanh Tho, Ph.D.

CSE – HCMUT

[qttho@cse.hcmut.edu.vn](mailto:qttho@cse.hcmut.edu.vn)

# Interfaces

The slide features a decorative header with six circles arranged in a horizontal line. The first two circles are positioned behind the title 'Interfaces'. The sequence of colors for the six circles is: solid light purple, hollow light purple, solid light purple, hollow light purple, hollow light purple, and solid light purple.

- What is an Interface?
- Creating an Interface
- Implementing an Interface
- What is Marker Interface?

# Inheritance

The title 'Inheritance' is positioned at the top left. To its right and below it are five circles arranged in a horizontal row. The first circle is solid light purple. The second circle is white with a light purple outline. The third circle is solid light purple. The fourth circle is white with a light purple outline. The fifth circle is solid light purple.

- OOP allows you to derive new classes from existing classes. This is called *inheritance*.
- Inheritance is an important and powerful concept in Java. Every class you define in Java is inherited from an existing class.
- Sometimes it is necessary to derive a subclass from several classes, thus inheriting their data and methods. In Java only one parent class is allowed.
- With *interfaces*, you can obtain effect of multiple inheritance.

# What is an Interface?

- An interface is a classlike construct that contains only constants and abstract methods.
- Cannot be instantiated. Only classes that implements interfaces can be instantiated.  
However, `final public static` variables can be defined to interface types.
- Why not just use abstract classes? Java does not permit multiple inheritance from classes, but permits implementation of *multiple interfaces*.

# What is an Interface?

- Protocol for classes that completely separates specification/behaviour from implementation.
- One class can implement many interfaces
- One interface can be implemented by many classes
- By providing the `interface` keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

# Why an Interface?

- Normally, in order for a method to be called from one class to another, both classes need to be present at *compile time* so the Java compiler can check to ensure that the method signatures are compatible.
- In a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses.
- Interfaces are designed to avoid this problem.
- Interfaces are designed to support dynamic method resolution at *run time*.

# Why an Interface?

- Interfaces are designed to avoid this problem.
- They disconnect the definition of a method or set of methods from the inheritance hierarchy.
- Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface.
- This is where the real power of interfaces is realized.

# Creating an Interface

File InterfaceName.java:

```
modifier interface InterfaceName  
{  
    constants declarations;  
    methods signatures;  
}
```

Modifier is **public** or not used.

File ClassName.java:

```
modifier Class ClassName implements InterfaceName  
{  
    methods implementation;  
}
```

If a class implements an interface, it *should override* all the abstract methods declared in the interface.



# Creating an Interface

```
public interface MyInterface
{
    public void aMethod1(int i);    // an abstract methods
    public void aMethod2(double a);
    ...
    public void aMethodN();
}
```

```
public Class MyClass implements MyInterface
{
    public void aMethod1(int i) { // implementaion }
    public void aMethod2(double a) { // implementaion }
    ...
    public void aMethodN() { // implementaion }
}
```

# Creating a Multiple Interface

```
modifier interface InterfaceName1 {  
    methods1 signatures;  
}  
  
modifier interface InterfaceName2 {  
    methods2 signatures;  
}  
  
...  
  
modifier interface InterfaceNameN {  
    methodsN signatures;  
}
```

# Creating a Multiple Interface

If a class implements a multiple interfaces, it *should override* all the abstract methods declared in all the interfaces.

```
public Class ClassName implements InterfaceName1,  
                                   InterfaceName2, ..., InterfaceNameN  
{  
    methods1 implementation;  
    methods2 implementation;  
    ...  
    methodsN implementation;  
}
```

# Example of Creating an Interface

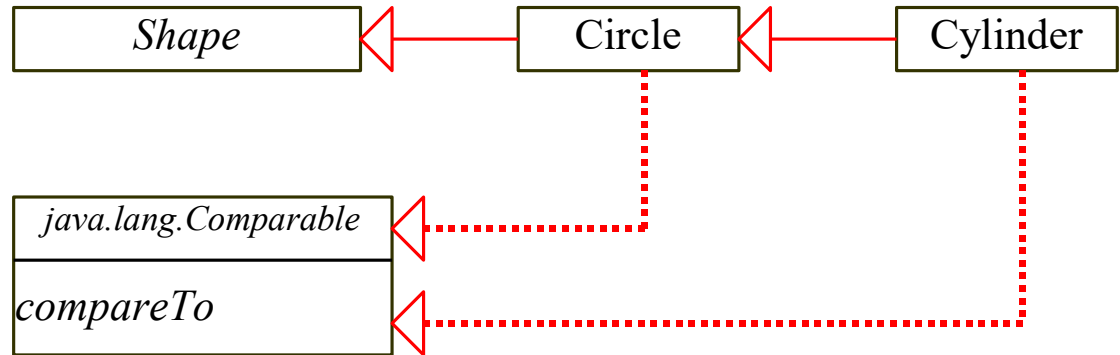
```
// This interface is defined in  
// java.lang package  
public interface Comparable  
{  
    public int compareTo(Object obj);  
}
```

# UML Notation

*UML Notation:*

*The interface name and the method names are italicized.*

*The dashed lines and hollow triangles are used to point to the interface.*



# Comparable Circle

```
public interface Comparable
{
    public int compareTo(Object obj);
}
```

---

```
public Class Circle extends Shape
                                implements Comparable {
    public int compareTo(Object o) {
        if (getRadius() > ((Circle)o).getRadius())
            return 1;
        else if (getRadius() < ((Circle)o).getRadius())
            return -1;
        else
            return 0;
    }
}
```

# Comparable Cylinder

```
public Class Cylinder extends Circle
    implements Comparable {
    public int compareTo(Object o) {
        if(findVolume() > ((Cylinder)o).findVolume())
            return 1;
        else if(findVolume() < ((Cylinder)o).findVolume())
            return -1;
        else
            return 0;
    }
}
```

# Generic findMax Method

```
public class Max
{
    // Return the maximum between two objects
    public static Comparable findMax(Comparable ob1,
                                     Comparable ob2)
    {
        if (ob1.compareTo(ob2) > 0)
            return ob1;
        else
            return ob2;
    }
}
```





# Access via interface reference

- You can declare variables as object references that use an *interface* rather than a class *type*. Any instance of any class that implements the declared interface can be stored in such a variable.

Example: `Comparable circle;`

- When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to.

Example: `circle = Max.findMax(circle1, circle2);`



# Using Interface

This is one of the key features of interfaces.

- The method to be executed is looked up dynamically at *run time*, allowing classes to be created later than the code which calls methods on them.
- The calling code can dispatch through an interface without having to know anything about the "callee."



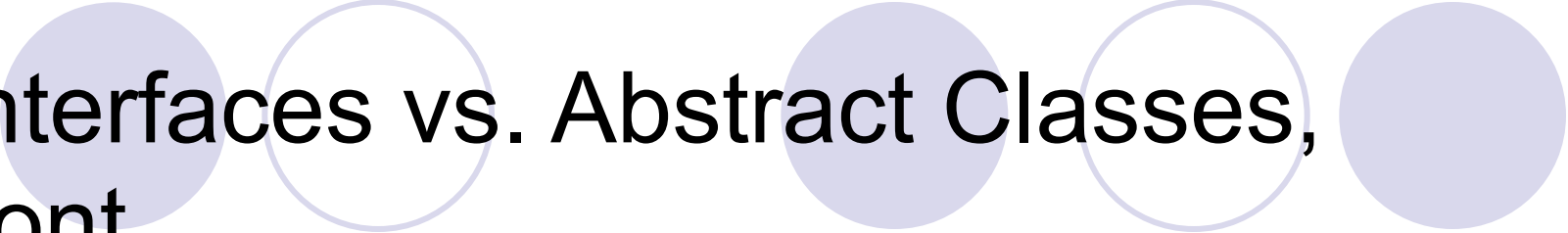
# Using Interface

Objective: Use the `findMax()` method to find a maximum circle between two circles:

```
Circle circle1 = new Circle(5);  
Circle circle2 = new Circle(10);  
Comparable circle = Max.findMax(circle1, circle2);  
System.out.println(((Circle)circle).getRadius());  
// 10.0  
System.out.println(circle.toString());  
// [Circle] radius = 10.0
```

# Interfaces vs. Abstract Classes

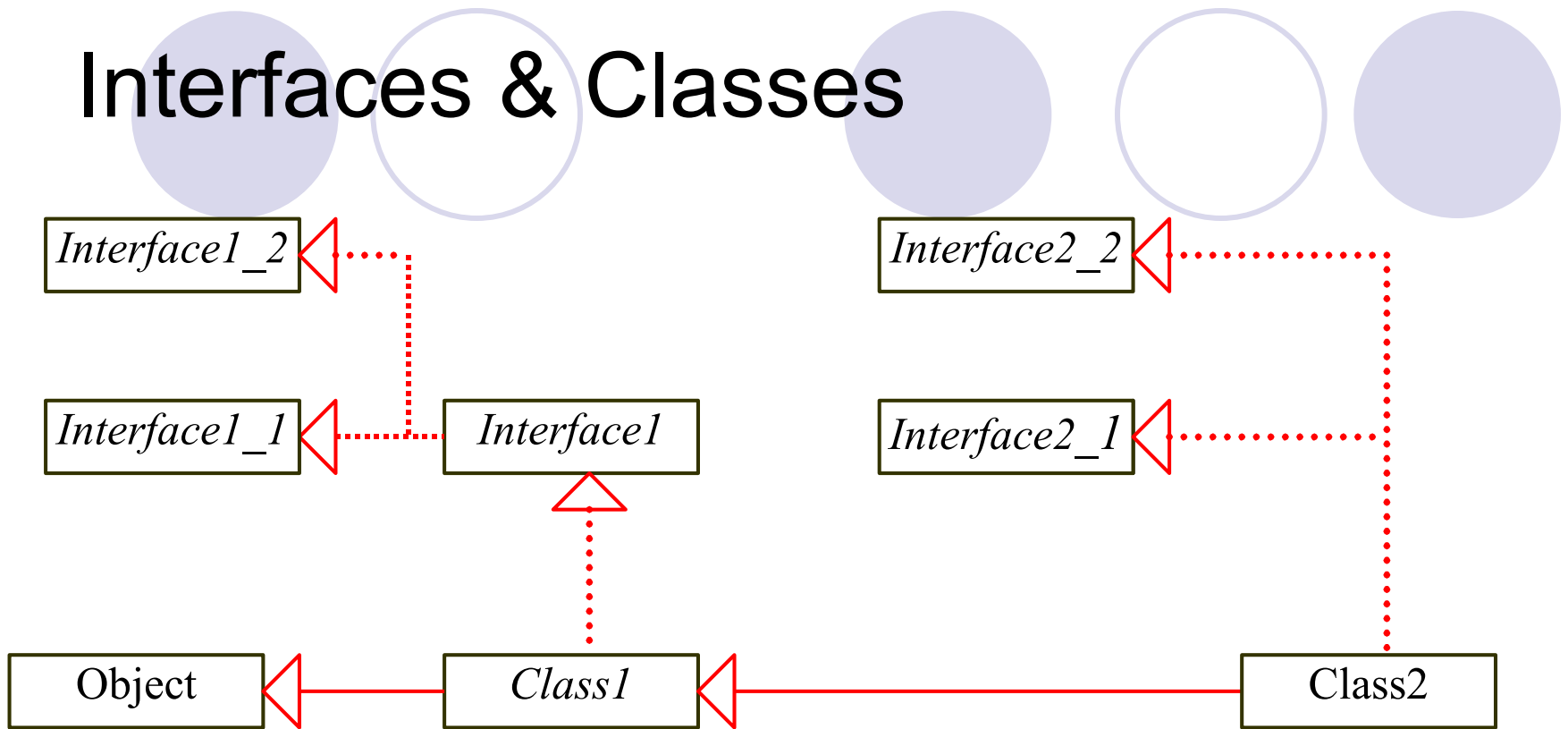
- ◆ In an interface, the data must be constants;  
an abstract class can have all types of data.
- ◆ Each method in an interface has only a signature without implementation;
- ◆ An abstract class can have concrete methods. An abstract class must contain at least one abstract method or inherit from another abstract method.



# Interfaces vs. Abstract Classes, cont.

Since all the methods defined in an interface are abstract methods, Java does not require you to put the `abstract` modifier in the methods in an interface, but you must put the `abstract` modifier before an abstract method in an abstract class.

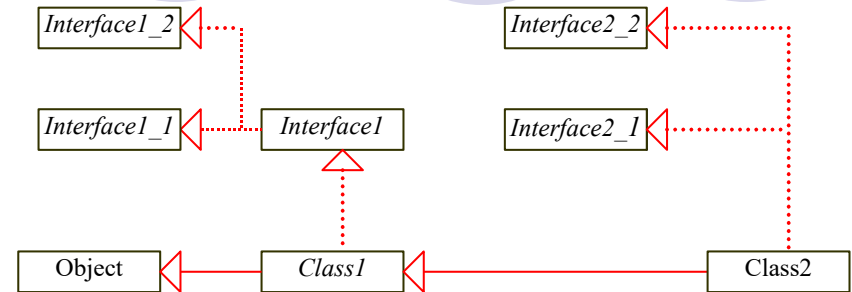
# Interfaces & Classes



- One interface can inherit another by use of the keyword `extends`. The syntax is the same as for inheriting classes.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

# Interfaces & Classes, cont.

```
public Interface1_1 { ... }  
public Interface1_2 { ... }  
public Interface2_1 { ... }  
public Interface2_2 { ... }
```



```
public Interface1 extends Interface1_1, Interface1_2  
{...}
```

```
public abstract Class Class1 implements Interface1  
{... }
```

```
public Class Class2 extends Class1 implements  
Interface2_1, Interface2_2 {...}
```

# Interfaces vs. Abstract Classes

- ◆ A *strong is-a* relationship that clearly describes a parent-child relationship should be modeled using classes.

Example: *a staff member is a person.*

- ◆ A *weak is-a* relationship, also known as *a-kind-of relationship*, indicates that an object possesses a certain *property*.

Example: all strings are comparable, so the `String` class implements the `Comparable` interface.

- ◆ The interface names are usually adjectives. You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired. You have to design one as a superclass, and the others as interfaces.



# The Cloneable Interfaces

Marker Interface: An empty interface.

A marker interface does not contain constants or methods, but it has a special meaning to the Java system. The Java system requires a class to implement the Cloneable interface to become

```
cloneable.  
public interface Cloneable  
{  
  
}
```

# The Cloneable Interfaces

```
public Class Circle extends Shape
                        implements Cloneable {
    public Object clone() {
        try {
            return super.clone()
        }
        catch (CloneNotSupportedException ex) {
            return null;
        }
    }
}
```

---

```
Circle c1 = new Circle(1,2,3);
Circle c2 = (Circle)c1.clone();
```

This is *shallow* copying with `super.clone()` method.  
Override the method for *deep* copying.