

Abstract Classes and Interfaces

Interfaces

The Comparable Interface

Interfaces vs. Abstract Classes

Interfaces

What is an interface?

Why is an interface useful?

How do you define an interface?

How do you use an interface?

What is an interface?

Why is an interface useful?

An interface is a class like construct that contains only constants and abstract methods. In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify a behavior for objects. For example, you can specify that the objects are comparable using the appropriate interface.

Define an Interface

To distinguish an interface from a class, Java uses the following syntax to define an interface:

```
public interface InterfaceName {  
    constant declarations;  
    method signatures;  
}
```

Example:

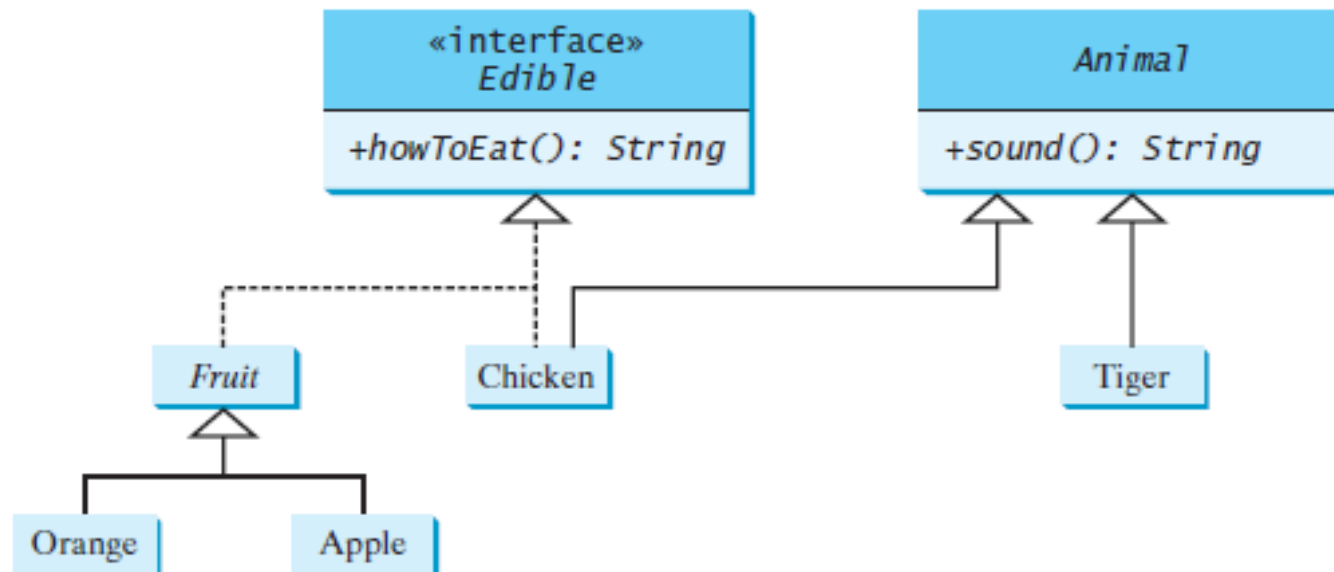
```
public interface Edible {  
    /** Describe how to eat */  
    public abstract String howToEat();  
}
```

Interface is a Special Class

An interface is treated like a special class in Java. Like an abstract class, you cannot create an instance from an interface using the new operator, but in most cases you can use an interface more or less the same way you use an abstract class. For example, you can use an interface as a data type for a variable.

Example

You can create the Edible interface and use it to specify whether an object is edible (i.e. can be eaten). This is accomplished by letting the class for the object implement this interface using the implements keyword. For example, the classes Chicken and Fruit implement the Edible interface. Note that in the UML diagram the name of the interface is italicized and is preceded by <<interface>>. Also the name of the abstract methods inside the interface are italicized. A dashed line is used to represent the “implements” relationship between the class and the interface. This relationship between the class and the interface is known as *interface inheritance*.



Example

```
public interface Edible {
    public abstract String howToEat();
}
abstract class Animal {
}
class Chicken extends Animal implements Edible {
    public String howToEat() {
        return "Chicken: Fry it";
    }
}
class Tiger extends Animal {
}
abstract class Fruit implements Edible {
}
class Apple extends Fruit {
    public String howToEat() {
        return "Apple: Make apple cider";
    }
}
class Orange extends Fruit {
    public String howToEat() {
        return "Orange: Make orange juice";
    }
}
```

Example

```
public class TestEdible {  
    public static void main(String[] args) {  
        Edible [] edibles = {new Orange(), new Chicken(), new Apple()};  
        for (int i = 0; i < edibles.length; i++) {  
            // if (edibles[i] instanceof Edible)  
                System.out.println(edibles[i].howToEat());  
        }  
    }  
}
```

OUTPUT

Orange: Make orange juice
Chicken: Fry it
Apple: Make apple cider

Omitting Modifiers in Interfaces

All data fields are *public final static* and all methods are *public abstract* in an interface. For this reason, these modifiers can be omitted, as shown below:

```
public interface T1 {  
    public static final int K = 1;  
  
    public abstract void p();  
}
```

Equivalent

```
public interface T1 {  
    int K = 1;  
  
    void p();  
}
```

A constant defined in an interface can be accessed using syntax InterfaceName.CONSTANT NAME (e.g., T1.K) since it is static.

The Comparable Interface

The **Comparable** interface, which is found in the java library, defines the **compareTo** method for comparing objects.

```
// This interface is defined in the java.lang package

package java.lang;

public interface Comparable<E>{
    public int compareTo(E o);
}
```

Defining Classes to Implement Comparable

```
public class Circle implements Comparable<Circle> {
    private double radius;
    public Circle(double r) {
        this.radius = r;
    }
    public double getRadius() {
        return radius;
    }
    public void setRadius(double radius) {
        this.radius = radius;
    }
    @Override // This method implements the compareTo method defined in the Comparable Interface
    public int compareTo(Circle o) {
        if (radius > o.radius)
            return 1;
        else if (radius < o.radius)
            return -1;
        else // case of radius == o.radius
            return 0;
    }
} // end class
```

Application to Test the Comparable Circle

```
public class Application {  
    public static void main(String[] args) {  
        Circle c1 = new Circle(1);  
        Circle c2 = new Circle(2);  
        Circle c3 = new Circle(2);  
        int x = c1.compareTo(c2);  
        System.out.println("Comparing c1 and c2:");  
        if (x == 0)  
            System.out.println("The two circles are equal.");  
        else if (x < 0)  
            System.out.println("The first circle is smaller.");  
        else // x > 0  
            System.out.println("The first circle is larger.");  
        System.out.println("Comparing c2 and c3:");  
        if (c2.compareTo(c3) == 0)  
            System.out.println("The two circles are equal.");  
        else if (c2.compareTo(c3) < 0)  
            System.out.println("The first circle is smaller.");  
        else // > 0  
            System.out.println("The first circle is larger.");  
    }  
}
```

OUTPUT

Comparing c1 and c2:
The first circle is smaller.
Comparing c2 and c3:
The two circles are equal.

Interfaces vs. Abstract Classes

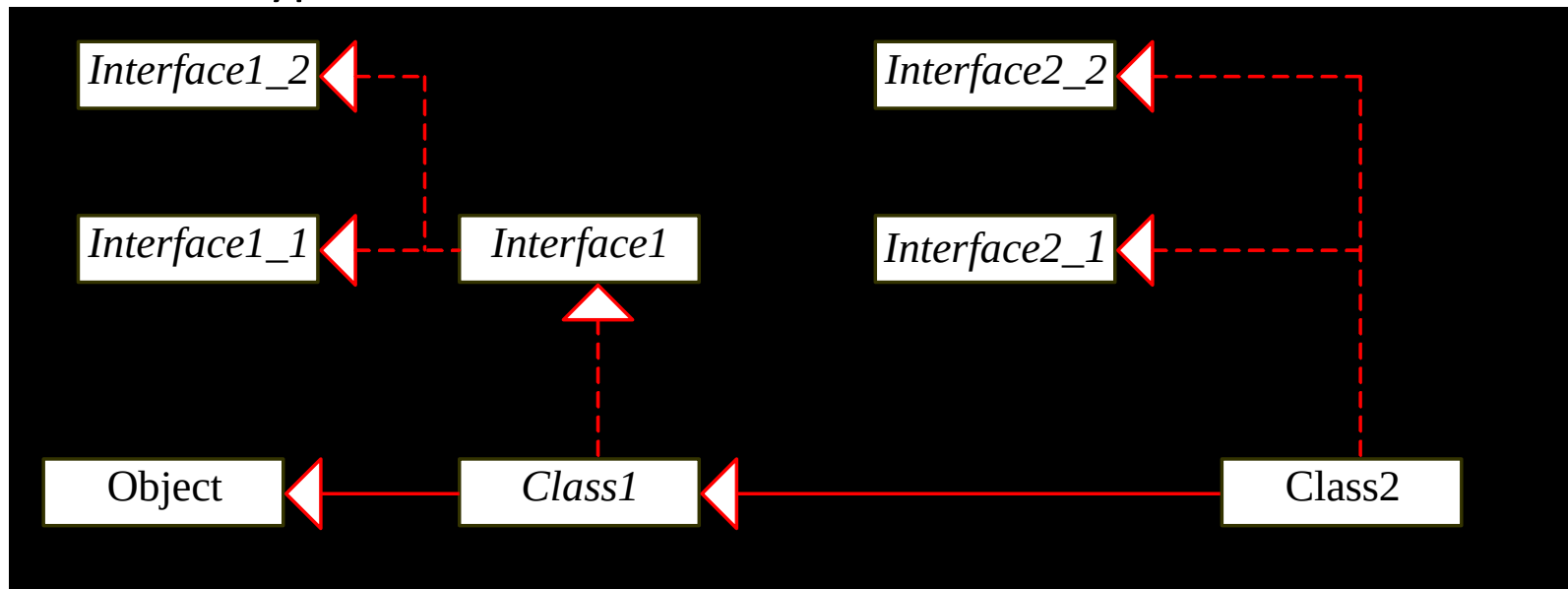
In an interface, the data must be constants; an abstract class can have all types of data.

Each method in an interface must be abstract ; an abstract class can have both abstract and concrete methods.

	Variables	Constructors	Methods
Abstract class	No restrictions	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be <u>public</u> <u>static</u> <u>final</u>	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

Interfaces vs. Abstract Classes, cont.

All classes share a single root, the Object class, but there is no single root for interfaces. Like a class, an interface also defines a type. A variable of an interface type can reference any instance of the class that implements the interface. If a class extends an interface, this interface plays the same role as a superclass. You can use an interface as a data type and cast a variable of an interface type to its subclass.



Suppose that *c* is an instance of **Class2**. *c* is also an instance of **Object**, **Class1**, **Interface1**, **Interface1_1**, **Interface1_2**, **Interface2_1**, and **Interface2_2**.

Interfaces vs. Abstract Classes, cont.

Java allows only single inheritance for class extension but allows multiple extensions for interfaces. For example,

```
public class A extends B implements Interface1, Interface2,  
Interface3 {  
    . . .  
}
```

An interface can inherit other interfaces using the `extends` keyword. Such an interface is called a subinterface. For example, `NewInterface` in the following code is a subinterface of `Interface1`, . . . , and `InterfaceN`.

```
public interface NewInterface extends Interface1, . . . ,  
InterfaceN {  
    . . .  
}
```

Interfaces vs. Abstract Classes, cont.

A class implementing **NewInterface** must implement the abstract methods defined in **NewInterface**, **Interface1**, . . . , and **InterfaceN**.

An interface can extend other interfaces but not classes.

A class can extend one superclass and implement multiple interfaces.

Whether to use an interface or an abstract class?

Abstract classes and interfaces can both be used to model common features. How do you decide whether to use an interface or an abstract class? In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes. For example, a staff member is a person. So their relationship should be modeled using class inheritance. A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces. For example, all strings are comparable, so the String class implements the Comparable interface. You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired. In the case of multiple inheritance, you have to design one as a superclass, and others as interface.