Interfaces

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?

- An interface is class-like construct for defining common operations for objects.
- An interface is a blueprint of a class. It has ***static constants*** and ***abstract methods***.
- An interface is similar to an abstract class, but the intent of an interface is to specify common behavior for objects. For example, you can specify that the objects are comparable, edible, cloneable using appropriate interfaces.

Interfaces in Java

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
- A Java library example is, Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

Define an Interface

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

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

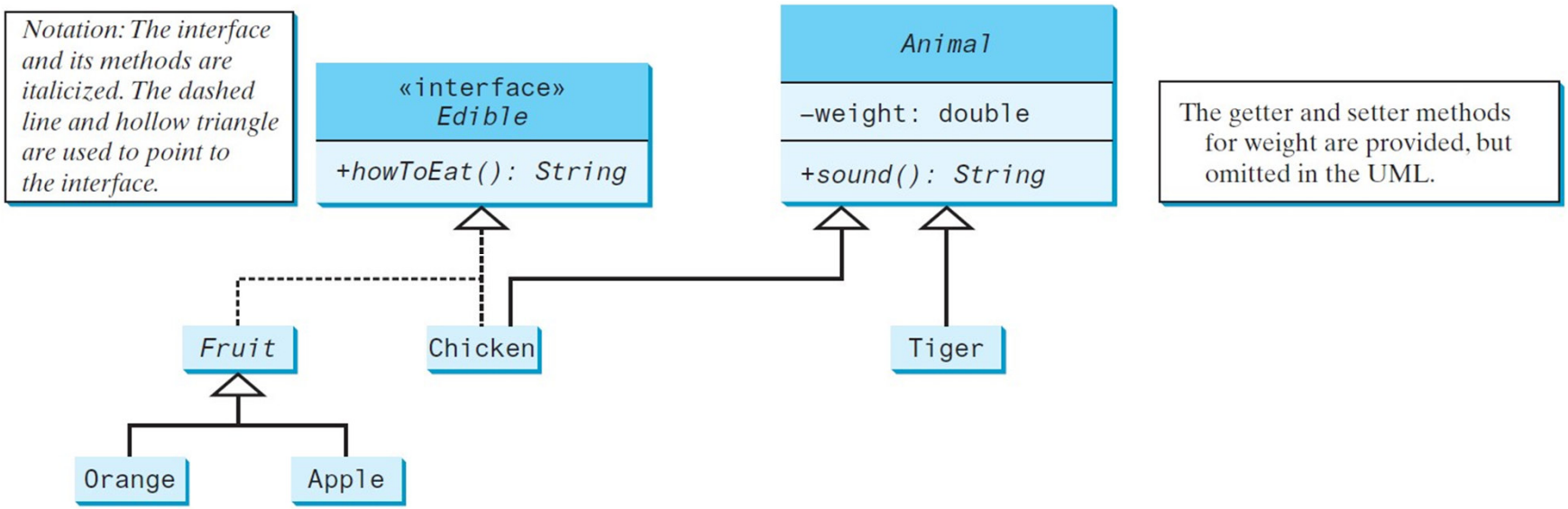
- To declare an interface, use **`interface`** keyword.

Interface is a special class

- An interface is treated as like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class.
- Like an abstract class you can't create an instance from an interface using the new operator, but in most of 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, as the result of casting, and so on.
- As with an abstract class, you cannot create an instance from an interface using the `new` operator.

Example

- You can use the **`Edible`** interface to specify whether an object is edible. 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.



Example : Edible; Fruit; Animal

```
public abstract class Fruit implements Edible {
// Data fields, constructors, and methods omitted here
}
```

```
public interface Edible {

/** Describe how to eat */
public abstract String  howToEat();

}
```

```
public abstract class Animal {
private double weight;

public double getweight() {
return this.weight;
}

public void setWeight(double weight) {
this.weight = weight;
}

/** return animal sound */
public abstract String sound();
}
```

Example: Apple; Orange;

```
public abstract class Fruit implements Edible {
// Data fields, constructors, and methods omitted here
}
```

```
public class Apple extends Fruit {

@Override
public String howToEat() {
return "Apple: Make apple cider";
}

}
```

```
public class Orange extends Fruit {

@Override
public String howToEat() {
return "Orange: Make orange juice";
}

}
```

Example: Chicken, Tiger

```
public class Chicken extends Animal implements Edible {
    @Override
    public String howToEat() {
        return "Chicken: Fry it";
    }

    @Override
    public String sound() {
        return "Chicken: cock-a-doodle-doo";
    }
}
```

```
public class Tiger extends Animal {
    @Override
    public String sound() {
        return "Tiger: RROOAARR";
    }
}
```

Example:

```
public class TestEdible{
    public static void main(String[] args) {

        Object[] objects = {new Tiger(), new Chicken(), new Apple()};

        for (int i = 0; i < objects.length; i++) {
            if (objects[i] instanceof Edible)
                System.out.println(((Edible)objects[i]).howToEat());

            if (objects[i] instanceof Animal)
                System.out.println(((Animal)objects[i]).sound());
        }
    }
}
```

RUN

Omitting Modifiers in Interfaces

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

```
public interface T {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T {
  int K = 1;

  void p();
}
```

- A constant defined in an interface can be accessed using syntax `InterfaceName.CONSTANT_NAME` (e.g. `T1.K`)

Example: The `Comparable` Interface

- The **`Comparable`** interface defines the **`compareTo`** method for comparing objects.

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

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

- The **`compareTo`** method determines the order of this object with the specified object **`o`** and returns a negative integer, zero, or a positive integer if this object is *less than, equal to*, or *greater than* **`o`**.

Example: The `Comparable` Interface

- The Comparable interface is a generic interface. The generic type E is replaced by a concrete type when implementing this interface.
- Since all the numeric wrapper classes and the Character class implement the Comparable interface, the `compareTo` method is implemented in these classes.

```java
public final class Integer extends Number
    implements Comparable<Integer> {
  // class body omitted

  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}
```

```java
public class BigInteger extends Number
    implements Comparable<Biginteger> {
  // class body omitted

  @Override
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}
```

```java
public final class String extends Object
    implements Comparable<String> {
  // class body omitted

  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}
```

```java
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

  @Override
  public int compareTo(Date o) {
    // Implementation omitted
  }
}
```

Examples

- Thus, numbers are comparable, strings are comparable, and so are dates. You can use the `compareTo` method to compare two numbers, two strings, and two dates. For example, the following code:

```
1. System.out.println(new Integer(4).compareTo(new Integer(25)));

2. System.out.println("ABC".compareTo("ABE"));

3. java.util.Date date1 = new java.util.Date(2020, 3, 1);

4. java.util.Date date2 = new java.util.Date(2019, 5, 8);

5. System.out.println(date1.compareTo(date2));
```

Generic `CompareTo` Method

- Let **n** be an **Integer** object, **s** be a **String** object, and **d** be a **Date** object. All the following expressions are true

```
n instanceof Integer
n instanceof Object
n instanceof Comparable
```

```
b instanceof String
b instanceof Object
b instanceof Comparable
```

```
d instanceof java.util.Date
d instanceof Object
d instanceof Comparable
```

- Since all **Comparable** objects have the `compareTo` method, the `java.util.Arrays.sort(Object[])` method in the Java API uses the `compareTo` method to compare and sorts the objects in an array, provided the objects are instances of the **Comparable** interface.

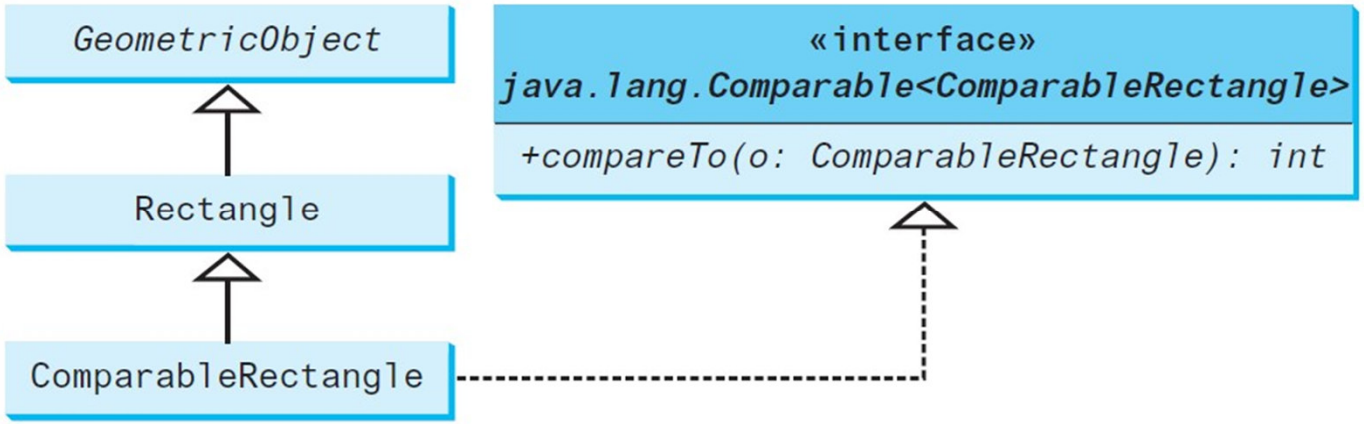
Example: SortComparableObjects

```
import java.math.BigInteger;

public class SortComparableObjects {

    public static void main(String[] args) {
        String[] cities = {"Savannah", "Boston", "Atlanta", "Tampa"};
        java.util.Arrays.sort(cities);
        for (String city: cities)
            System.out.print(city + " ");
        System.out.println();

    BigInteger[] hugeNumbers = {
            new BigInteger("323454543534534535"),
            new BigInteger("987229072984248243262"),
            new BigInteger("164132646563216264546")};
    java.util.Arrays.sort(hugeNumbers);
    for(BigInteger number: hugeNumbers)
        System.out.print(number + "; ");
    }
}
```

RUN

Defining Classes to Implement comparable

Notation:
The interface name and the method names are italicized. The dashed line and hollow triangle are used to point to the interface.



Example: ComparableRectangle

SortRetangles.java | ComparableRectangle.java | Rectangle.java | GeometricObject.java

```
1
2  public class ComparableRectangle extends Rectangle implements Comparable<ComparableRectangle> {
3
4      public ComparableRectangle() {
5          super();
6      }
7
8      public ComparableRectangle(double width, double height) {
9          super(width, height);
10     }
11
12     @Override
13     public int compareTo(ComparableRectangle o) {
14         if (getArea() > o.getArea())
15             return 1;
16         else if (getArea() < o.getArea())
17             return -1;
18         else
19             return 0;
20     }
21
22     @Override
23     public String toString() {
24         return super.toString() + " Area: " + getArea();
25     }
26
27 }
```

Example: SortRectangles

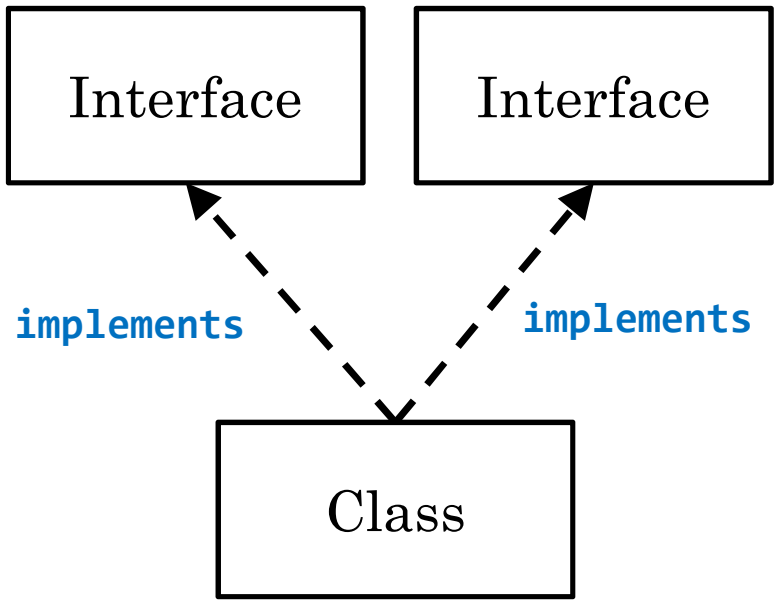
SortRetangles.java | ComparableRectangle.java

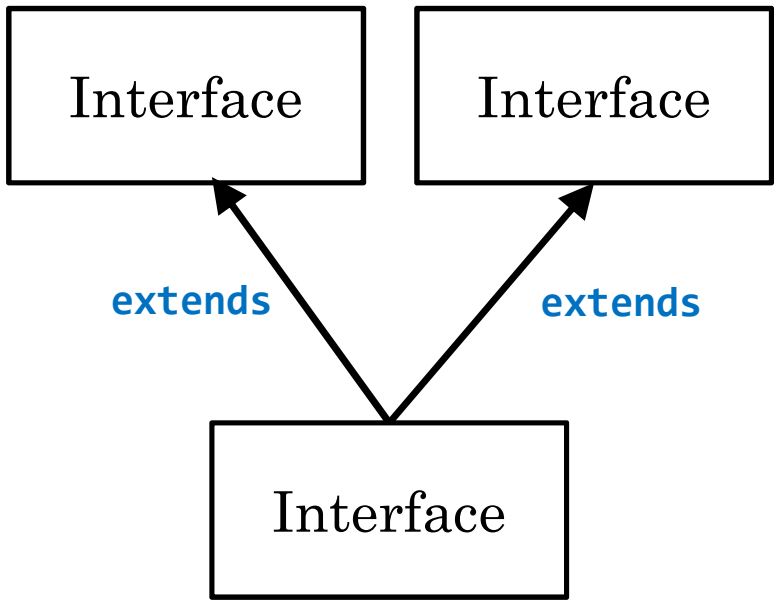
```
1
2  public class SortRetangles {
3      public static void main(String[] args) {
4
5          System.out.println("SortRetangles Demo output: ");
6          System.out.println("---------------------------------------");
7
8          ComparableRectangle [] rectangles = {
9                  new ComparableRectangle(5.4, 3.1),
10                 new ComparableRectangle(4.4, 6.2),
11                 new ComparableRectangle(7.4, 3.8),
12                 new ComparableRectangle(2.4, 8.1),
13         };
14
15         java.util.Arrays.sort(rectangles);
16         for (Rectangle rectangle: rectangles) {
17             System.out.println(rectangle + " ");
18             System.out.println();
19         }
20
21         System.out.println("---------------------------------------");
22         System.out.println();
23     }
24 }
```

RUN

Implementing multiple interfaces

- Unlike inheritance which allows each class to have only one parent, you may design a class that implements more than one interface.





Example: GeometricObject Class

- Say, we need two additional features for the **GeometricObject** Class: **PenWidth** and **Diagonal**

```
public interface PenWidth {
    int THIN = 1, THICK = 8;
    void setPenWidth(int w);
    int getPenWidth();
}
```

```
public interface Diagonal {
    double getDiagonal();
}
```

Example (Cont'd)

- We declare a subclass of **`Rectangle`**– **`OutlineRectangle`** that implements two interfaces while retaining the original features of **`Rectangle`**

```
public class OutlineRectangle extends Rectangle implements PenWidth, Diagonal {

    private int pw = PenWidth.THIN;
    @Override
    public double getDiagonal() {
        double w = getWidth(), h = getHeight();
        return Math.sqrt(w*w + h*h);
    }
    @Override
    public void setPenWidth(int w) { pw = w; }
    @Override
    public int getPenWidth() { return pw; }

}
```

Example: using interface as data type

- We write a method to set the pen width of all objects that implement **PenWidth**

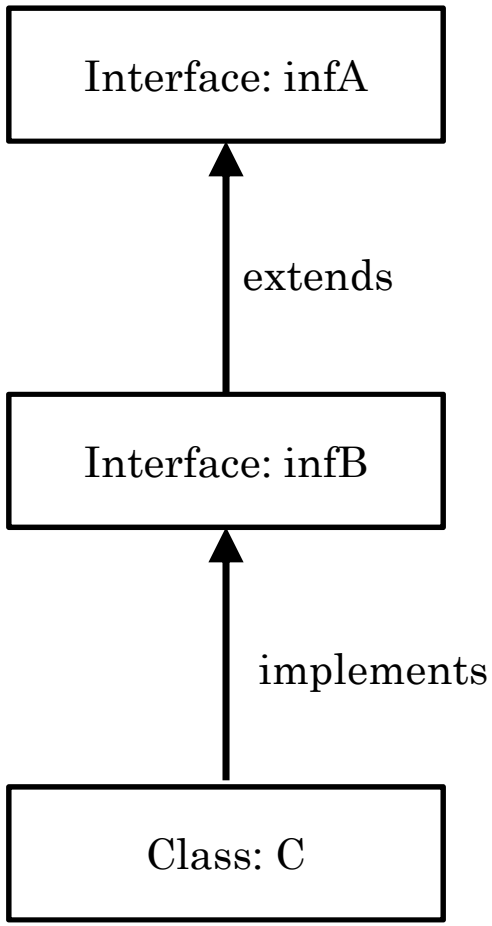
```
public static void setAllPenWidths(PenWidth[] a, int w) {
    for (PenWidth x : a) x.setPenWidth(w);
}
```

- We can pass an array of outlined rectangles to **setAllPenWidths**

```
OutlineRectangle[] a = {new OutlineRectangle(5.4, 3.1),
                        new OutlineRectangle(4.4, 6.2),
                        new OutlineRectangle(7.4, 3.8),
                        new OutlineRectangle(2.4, 8.1),};

setAllPenWidths(a, PenWidth.THICK);
```

Interface Inheritance

```
interface infA {
    void m1();
}

interface infB extends infA {
    void m2();
}


class C implements infB {
    @Override
    public void m1() {
        System.out.println("implement method m1");
}
    @Override
    public void m2() {
        System.out.println("implement method m2");
    }
}
```

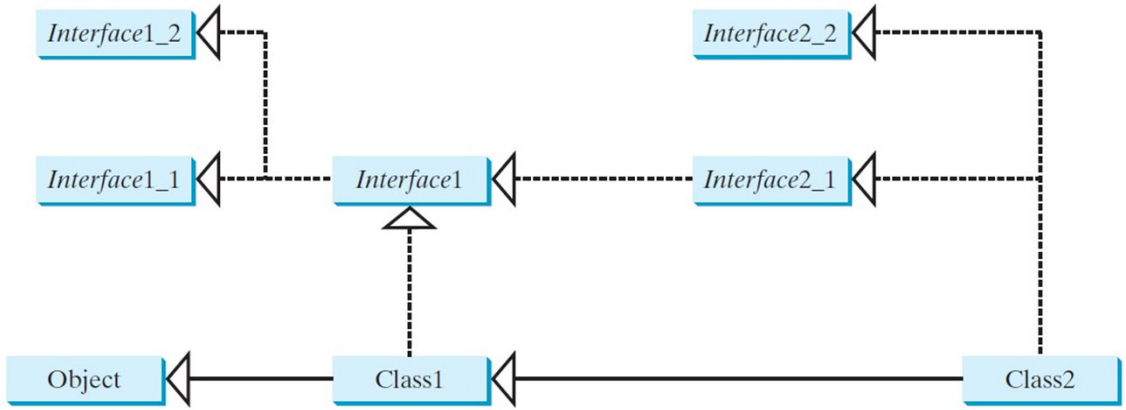


Interfaces vs. Abstract Classes

	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 `public static final`.	No constructors. An interface cannot be instantiated using the new operator.	May contain public abstract instance methods, public default (Java 8), and public static methods (Java 8).

Interfacd 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 implements an interface, the interface is like a superclass for the class. You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa.
- For example, suppose **c** is an instance of Class2. **c** is also an instance of Object, Class1, Interface1,Interface1_1, Interface1_2, Interface2_1, and Interface2_2.



Caution: conflict interfaces

- In rare occasions, a class may implement two interfaces with conflict information (e.g. two same constants with different values or two methods with same signature but different return type). This type of error will be detected by the compiler.

Whether to use an interface or a class?

- In general, a strong is-a relationship that clearly describes a parent–child relationship should be modeled using classes. Abstract class for what a class is.
- 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. Interface for what a class can do.
- You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired.