Implementing an Interface

Defining the Interface Relatable

To declare a class that implements an interface, you include an `implements` clause in the class declaration. Your class can implement more than one interface, so the `implements` keyword is followed by a comma-separated list of the interfaces implemented by the class. By convention, the `implements` clause follows the `extends` clause, if there is one.

Consider an interface that defines how to compare the size of objects.

```
public interface Relatable {

    // this (object calling isLargerThan())
    // and other must be instances of
    // the same class returns 1, 0, -1
    // if this is greater than,
    // equal to, or less than other
    public int isLargerThan(Relatable other);
}
```

If you want to be able to compare the size of similar objects, no matter what they are, the class that instantiates them should implement `Relatable`.

Any class can implement `Relatable` if there is some way to compare the relative "size" of objects instantiated from the class. For strings, it could be number of characters; for books, it could be number of pages; for students, it could be weight; and so forth. For planar geometric objects, area would be a good choice (see the `RectanglePlus` class that follows), while volume would work for three-dimensional geometric objects. All such classes can implement the `isLargerThan()` method.

If you know that a class implements `Relatable`, then you know that you can compare the size of the objects instantiated from that class.

Implementing the Relatable Interface

Here is the `Rectangle` class that was presented in the Creating Objects section, rewritten to implement `Relatable`.

```
public class RectanglePlus
    implements Relatable {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // four constructors
    public RectanglePlus() {
        origin = new Point(0, 0);
    }
    public RectanglePlus(Point p) {
        origin = p;
    }
    public RectanglePlus(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
    public RectanglePlus(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }

    // a method for moving the rectangle
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }

    // a method for computing
    // the area of the rectangle
    public int getArea() {
        return width * height;
    }

    // a method required to implement
    // the Relatable interface
    public int isLargerThan(Relatable other) {
        RectanglePlus otherRect
            = (RectanglePlus)other;
        if (this.getArea() < otherRect.getArea())
            return -1;
        else if (this.getArea() > otherRect.getArea())
            return 1;
        else
            return 0;
    }
}
```

Because `RectanglePlus` implements `Relatable`, the size of any two `RectanglePlus` objects can be compared.

> *Note: The `isLargerThan()` method, as defined in the `Relatable` interface, takes an object of type `Relatable`. The line of code casts other to a `RectanglePlus` instance. Type casting tells the compiler what the object really is. Invoking `getArea()` directly on the other instance (`other.getArea()`) would fail to compile because the compiler does not understand that other is actually an instance of `RectanglePlus`.*

Evolving Interfaces

Consider an interface that you have developed called `DoIt`:

```
public interface DoIt {
   void doSomething(int i, double x);
   int doSomethingElse(String s);
}
```

Suppose that, at a later time, you want to add a third method to `DoIt`, so that the interface now becomes:

```
public interface DoIt {

   void doSomething(int i, double x);
   int doSomethingElse(String s);
   boolean didItWork(int i, double x, String s);

}
```

If you make this change, then all classes that implement the old `DoIt` interface will break because they no longer implement the old interface. Programmers relying on this interface will protest loudly.

Try to anticipate all uses for your interface and specify it completely from the beginning. If you want to add additional methods to an interface, you have several options. You could create a `DoItPlus` interface that extends `DoIt`:

```java
public interface DoItPlus extends DoIt {

   boolean didItWork(int i, double x, String s);

}
```

Now users of your code can choose to continue to use the old interface or to upgrade to the new interface.

Alternatively, you can define your new methods as default methods. The following example defines a default method named `didItWork()`:

```java
public interface DoIt {

   void doSomething(int i, double x);
   int doSomethingElse(String s);
   default boolean didItWork(int i, double x, String s) {
       // Method body
   }
}
```

Note that you must provide an implementation for default methods. You could also define new static methods to existing interfaces. Users who have classes that implement interfaces enhanced with new default or static methods do not have to modify or recompile them to accommodate the additional methods.

Default Methods

The section Interfaces describes an example that involves manufacturers of computer-controlled cars who publish industry-standard interfaces that describe which methods can be invoked to operate their cars. What if those computer-controlled car manufacturers add new functionality, such as flight, to their cars? These manufacturers would need to specify new methods to enable other companies (such as electronic guidance instrument manufacturers) to adapt their software to flying cars. Where would these car manufacturers declare these new flight-related methods? If they add them to their original interfaces, then programmers who have implemented those interfaces would have to rewrite their implementations. If they add them as static methods, then programmers would regard them as utility methods, not as essential, core methods.

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

Consider the following interface, `TimeClient`:

```java
import java.time.*;

public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
                               int hour, int minute, int second);
    LocalDateTime getLocalDateTime();
}
```

The following class, `SimpleTimeClient`, implements `TimeClient`:

```java
public class SimpleTimeClient implements TimeClient {

    private LocalDateTime dateAndTime;

    public SimpleTimeClient() {
        dateAndTime = LocalDateTime.now();
    }

    public void setTime(int hour, int minute, int second) {
        LocalDate currentDate = LocalDate.from(dateAndTime);
        LocalTime timeToSet = LocalTime.of(hour, minute, second);
        dateAndTime = LocalDateTime.of(currentDate, timeToSet);
    }

    public void setDate(int day, int month, int year) {
        LocalDate dateToSet = LocalDate.of(day, month, year);
        LocalTime currentTime = LocalTime.from(dateAndTime);
        dateAndTime = LocalDateTime.of(dateToSet, currentTime);
    }

    public void setDateAndTime(int day, int month, int year,
                               int hour, int minute, int second) {
        LocalDate dateToSet = LocalDate.of(day, month, year);
        LocalTime timeToSet = LocalTime.of(hour, minute, second);
```

```java
            dateAndTime = LocalDateTime.of(dateToSet, timeToSet);
        }

        public LocalDateTime getLocalDateTime() {
            return dateAndTime;
        }

        public String toString() {
            return dateAndTime.toString();
        }

        public static void main(String... args) {
            TimeClient myTimeClient = new SimpleTimeClient();
            System.out.println(myTimeClient.toString());
        }
    }
```

Suppose that you want to add new functionality to the `TimeClient` interface, such as the ability to specify a time zone through a `ZonedDateTime` object (which is like a `LocalDateTime` object except that it stores time zone information):

```java
public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
        int hour, int minute, int second);
    LocalDateTime getLocalDateTime();
    ZonedDateTime getZonedDateTime(String zoneString);
}
```

Following this modification to the `TimeClient` interface, you would also have to modify the class `SimpleTimeClient` and implement the method `getZonedDateTime()`. However, rather than leaving `getZonedDateTime()` as abstract (as in the previous example), you can instead define a default implementation. (Remember that an abstract method is a method declared without an implementation.)

```java
public interface TimeClient {
    void setTime(int hour, int minute, int second);
    void setDate(int day, int month, int year);
    void setDateAndTime(int day, int month, int year,
                               int hour, int minute, int second);
    LocalDateTime getLocalDateTime();

    static ZoneId getZoneId (String zoneString) {
        try {
            return ZoneId.of(zoneString);

        } catch (DateTimeException e) {
            System.err.println("Invalid time zone: " + zoneString +
                "; using default time zone instead.");
            return ZoneId.systemDefault();
        }
    }

    default ZonedDateTime getZonedDateTime(String zoneString) {
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
    }
}
```

You specify that a method definition in an interface is a default method with the `default` keyword at the beginning of the method signature. All method declarations in an interface, including default methods, are implicitly public, so you can omit the public modifier.

With this interface, you do not have to modify the class `SimpleTimeClient`, and this class (and any class that implements the interface `TimeClient`), will have the method `getZonedDateTime()` already defined. The following example, `TestSimpleTimeClient`, invokes the method `getZonedDateTime()` from an instance of `SimpleTimeClient`:

```java
public class TestSimpleTimeClient {
    public static void main(String... args) {
        TimeClient myTimeClient = new SimpleTimeClient();
        System.out.println("Current time: " + myTimeClient.toString());
        System.out.println("Time in California: " +
            myTimeClient.getZonedDateTime("Blah blah").toString());
    }
}
```

Extending Interfaces That Contain Default Methods

When you extend an interface that contains a default method, you can do the following:

- Not mention the default method at all, which lets your extended interface inherit the default method.
- Redeclare the default method, which makes it abstract.
- Redefine the default method, which overrides it.
- Suppose that you extend the interface `TimeClient` as follows:

```
public interface AnotherTimeClient extends TimeClient { }
```

Any class that implements the interface `AnotherTimeClient` will have the implementation specified by the default method `TimeClient.getZonedDateTime()`.

Suppose that you extend the interface `TimeClient` as follows:

```
public interface AbstractZoneTimeClient extends TimeClient {
    public ZonedDateTime getZonedDateTime(String zoneString);
}
```

Any class that implements the interface `AbstractZoneTimeClient` will have to implement the method `getZonedDateTime()`; this method is an abstract method like all other non-default (and non-static) methods in an interface.

Suppose that you extend the interface TimeClient as follows:

```
public interface HandleInvalidTimeZoneClient extends TimeClient {
    default public ZonedDateTime getZonedDateTime(String zoneString) {
        try {
            return ZonedDateTime.of(getLocalDateTime(),ZoneId.of(zoneString));
        } catch (DateTimeException e) {
            System.err.println("Invalid zone ID: " + zoneString +
                "; using the default time zone instead.");
            return ZonedDateTime.of(getLocalDateTime(),ZoneId.systemDefault());
        }
    }
}
```

Any class that implements the interface `HandleInvalidTimeZoneClient` will use the implementation of `getZonedDateTime()` specified by this interface instead of the one specified by the interface `TimeClient`.

Static Methods

In addition to default methods, you can define static methods in interfaces. (A static method is a method that is associated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods.) This makes it easier for you to organize helper methods in your libraries; you can keep static methods specific to an interface in the same interface rather than in a separate class. The following example defines a static method that retrieves a `ZoneId` object corresponding to a time zone identifier; it uses the system default time zone if there is no ZoneId object corresponding to the given identifier. (As a result, you can simplify the method getZonedDateTime()):

```java
public interface TimeClient {
    // ...
    static public ZoneId getZoneId (String zoneString) {
        try {
            return ZoneId.of(zoneString);
        } catch (DateTimeException e) {
            System.err.println("Invalid time zone: " + zoneString +
                "; using default time zone instead.");
            return ZoneId.systemDefault();
        }
    }

    default public ZonedDateTime getZonedDateTime(String zoneString) {
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
    }
}
```

Like static methods in classes, you specify that a method definition in an interface is a static method with the `static` keyword at the beginning of the method signature. All method declarations in an interface, including static methods, are implicitly public, so you can omit the public modifier.

Starting with Java SE 9, you can define private methods in a interface to abstract a common piece of code from default methods of an interface while defining its implementation. These methods belong to the implementation, and they can be neither default nor abstract when defined. For example, you could make `getZoneId` method private since it hosts a piece of code that is internal to the interface implementation.

Integrating Default Methods into Existing Libraries

Default methods enable you to add new functionality to existing interfaces and ensure binary compatibility with code written for older versions of those interfaces. In particular, default methods enable you to add methods that accept lambda expressions as parameters to existing interfaces. This section demonstrates how the `Comparator` interface has been enhanced with default and static methods.

Consider the `Card` and `Deck` classes. The `Card` interface contains two `enum` types (`Suit` and `Rank`) and two abstract methods (`getSuit()` and `getRank()`):

```java
public interface Card extends Comparable<Card> {

    public enum Suit {
        DIAMONDS (1, "Diamonds"),
        CLUBS    (2, "Clubs"   ),
        HEARTS   (3, "Hearts"  ),
        SPADES   (4, "Spades"  );

        private final int value;
        private final String text;
        Suit(int value, String text) {
            this.value = value;
            this.text = text;
        }
        public int value() {return value;}
        public String text() {return text;}
    }

    public enum Rank {
        DEUCE  (2 , "Two"  ),
        THREE  (3 , "Three"),
        FOUR   (4 , "Four" ),
        FIVE   (5 , "Five" ),
        SIX    (6 , "Six"  ),
        SEVEN  (7 , "Seven"),
        EIGHT  (8 , "Eight"),
        NINE   (9 , "Nine" ),
        TEN    (10, "Ten"  ),
        JACK   (11, "Jack" ),
        QUEEN  (12, "Queen"),
        KING   (13, "King" ),
        ACE    (14, "Ace"  );
        private final int value;
        private final String text;
        Rank(int value, String text) {
            this.value = value;
            this.text = text;
        }
        public int value() {return value;}
        public String text() {return text;}
    }

    public Card.Suit getSuit();
    public Card.Rank getRank();
}
```

The `Deck` interface contains various methods that manipulate cards in a deck:

```java
public interface Deck {

    List<Card> getCards();
    Deck deckFactory();
    int size();
    void addCard(Card card);
    void addCards(List<Card> cards);
    void addDeck(Deck deck);
    void shuffle();
    void sort();
    void sort(Comparator<Card> c);
    String deckToString();

    Map<Integer, Deck> deal(int players, int numberOfCards)
        throws IllegalArgumentException;

}
```

The class `PlayingCard` implements the interface `Card`, and the class `StandardDeck` implements the interface `Deck`.

```java
public class PlayingCard implements Card {

    private Rank rank;
    private Suit suit;

    // constructor

    // implementations of Card abstract methods
    public Suit getSuit() {
        return this.suit();
    }
    public Rank getRank() {
        return this.rank();
    }

    // implementation of Comparable<Card> method
    public int compareTo(Card o) {
        return this.hashCode() - o.hashCode();
    }

    // toString, equals, hashCode
}
```

The class `StandardDeck` implements the abstract method `Deck.sort()` as follows:

```java
public class StandardDeck implements Deck {

    private List<Card> entireDeck;

    // constructor, accessors

    // you need to add all the methods from Deck
    public void sort() {
        Collections.sort(entireDeck);
    }

    // toString, equals, hashCode
}
```

The method `Collections.sort()` sorts an instance of `List` whose element type implements the interface `Comparable`. The member `entireDeck` is an instance of `List` whose elements are of the type `Card`, which extends `Comparable`. The class `PlayingCard` implements the `Comparable.compareTo()` method as follows:

```java
public int hashCode() {
    return ((suit.value()-1)*13)+rank.value();
}

public int compareTo(Card o) {
    return this.hashCode() - o.hashCode();
}
```

The method `compareTo()` causes the method `StandardDeck.sort()` to sort the deck of cards first by suit, and then by rank.

What if you want to sort the deck first by rank, then by suit? You would need to implement the `Comparator` interface to specify new sorting criteria, and use the method `sort(List<T> list, Comparator<? super T> c)` (the version of the sort method that includes a `Comparator` parameter). You can define the following method in the class `StandardDeck`:

```java
public void sort(Comparator<Card> c) {
    Collections.sort(entireDeck, c);
}
```

With this method, you can specify how the method `Collections.sort()` sorts instances of the `Card` class. One way to do this is to implement the `Comparator` interface to specify how you want the cards sorted. The example `SortByRankThenSuit` does this:

```
public class SortByRankThenSuit implements Comparator<Card> {
    public int compare(Card firstCard, Card secondCard) {
        int compVal =
            firstCard.getRank().value() - secondCard.getRank().value();
        if (compVal != 0)
            return compVal;
        else
            return firstCard.getSuit().value() - secondCard.getSuit().value();
    }
}
```

The following invocation sorts the deck of playing cards first by rank, then by suit:

```
StandardDeck myDeck = new StandardDeck();
myDeck.shuffle();
myDeck.sort(new SortByRankThenSuit());
```

However, this approach is too verbose; it would be better if you could specify just the sort criteria and avoid creating multiple sorting implementations. Suppose that you are the developer who wrote the `Comparator` interface. What default or static methods could you add to the `Comparator` interface to enable other developers to more easily specify sort criteria?

To start, suppose that you want to sort the deck of playing cards by rank, regardless of suit. You can invoke the `StandardDeck.sort()` method as follows:

```
StandardDeck myDeck = new StandardDeck();
myDeck.shuffle();
myDeck.sort(
    (firstCard, secondCard) ->
        firstCard.getRank().value() - secondCard.getRank().value()
);
```

Because the interface `Comparator` is a functional interface, you can use a lambda expression as an argument for the `sort()` method. In this example, the lambda expression compares two integer values.

It would be simpler for your developers if they could create a `Comparator` instance by invoking the method `Card.getRank()` only. In particular, it would be helpful if your developers could create a `Comparator` instance that compares any object that can return a numerical value from a method such as `getValue()` or `hashCode()`. The `Comparator` interface has been enhanced with this ability with the static method comparing:

```
myDeck.sort(Comparator.comparing((card) -> card.getRank()));
```

In this example, you can use a method reference instead:

```
myDeck.sort(Comparator.comparing(Card::getRank));
```

This invocation better demonstrates how to specify different sort criteria and avoid creating multiple sorting implementations.

The `Comparator` interface has been enhanced with other versions of the static method comparing such as `comparingDouble()` and `comparingLong()` that enable you to create `Comparator` instances that compare other data types.

Suppose that your developers would like to create a `Comparator` instance that could compare objects with more than one criteria. For example, how would you sort the deck of playing cards first by rank, and then by suit? As before, you could use a lambda expression to specify these sort criteria:

```
StandardDeck myDeck = new StandardDeck();
myDeck.shuffle();
myDeck.sort(
    (firstCard, secondCard) -> {
        int compare =
            firstCard.getRank().value() - secondCard.getRank().value();
        if (compare != 0)
            return compare;
        else
            return firstCard.getSuit().value() - secondCard.getSuit().value();
    }
);
```

It would be simpler for your developers if they could build a `Comparator` instance from a series of `Comparator` instances. The `Comparator` interface has been enhanced with this ability with the default method `thenComparing()`:

```
myDeck.sort(
    Comparator
        .comparing(Card::getRank)
        .thenComparing(Comparator.comparing(Card::getSuit)));
```

The `Comparator` interface has been enhanced with other versions of the default method `thenComparing()`, such as `thenComparingDouble()` and `thenComparingLong()`, that enable you to build `Comparator` instances that compare other data types.

Suppose that your developers would like to create a `Comparator` instance that enables them to sort a collection of objects in reverse order. For example, how would you sort the deck of playing cards first by descending order of rank, from Ace to Two (instead of from Two to Ace)? As before, you could specify another lambda expression. However, it would be simpler for your developers if they could reverse an existing `Comparator` by invoking a method. The `Comparator` interface has been enhanced with this ability with the default method `reversed()`:

```
myDeck.sort(
    Comparator.comparing(Card::getRank)
        .reversed()
        .thenComparing(Comparator.comparing(Card::getSuit)));
```

This example demonstrates how the `Comparator` interface has been enhanced with default methods, static methods, lambda expressions, and method references to create more expressive library methods whose functionality programmers can quickly deduce by looking at how they are invoked. Use these constructs to enhance the interfaces in your libraries.

Last update: *September 14, 2021*

Interfaces ← **Implementing an Interface** → Using an Interface as a Type

Home > Tutorials > Interfaces > Implementing an Interface