<http://javarevisited.blogspot.com/2012/04/10-points-on-interface-in-java-with.html>

What is interface in Java with Example – Tutorial

**What is interface in Java**   
**Interface in java** is core part of Java programming language and one of the way to achieve [abstraction in Java](http://javarevisited.blogspot.sg/2010/10/abstraction-in-java.html) along with abstract class. Even though interface is fundamental [object oriented concept](http://javarevisited.blogspot.sg/2012/03/10-object-oriented-design-principles.html) ; Many Java programmers thinks Interface in Java as advanced concept and refrain using interface from early in programming career. At very basic level  **interface  in java** is a keyword  but same time it is an **object oriented term to define contracts** and **abstraction** , This contract is followed by any implementation of Interface in Java. Since [multiple inheritance is not allowed in Java](http://javarevisited.blogspot.com/2011/07/why-multiple-inheritances-are-not.html),  interface is only way to implement multiple inheritance at Type level. In this **Java tutorial** we will see **What is an interface in Java**, How to use interface in Java and where to use interface in Java and some important points related to Java interface. *What is an interface in Java* is also a [common core Java question](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) which people asked on various programming exams and interviews.

## Key Points about Interface in Java

In last section we saw What is an interface in Java and learned that interface provides abstraction in Java and its only way to achieve multiple inheritance at type level in Java. In this section we will see some important properties of interface in Java.  
  
1. Interface in java is declared using keyword interface and it represent a Type like any [Class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html). a reference variable of type interface can point to any implementation of that interface in Java. Its also a good [Object oriented design principle](http://javarevisited.blogspot.com/2012/03/10-object-oriented-design-principles.html) to "**program for interfaces than implementation**" because when you use interface to declare reference variable, method return type or method argument you are flexible enough to accept any future implementation of that interface which could be much better and high performance alternative of current implementation. similarly calling any method on interface doesn't tie you with any particular implementation and you can leverage  benefit of better or improved implementation over time. This maintenance aspect of interface is also sought in various [software design interview questions in Java](http://javarevisited.blogspot.sg/2012/06/20-design-pattern-and-software-design.html).

2) All variables declared inside interface is implicitly public [final variable](http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html) or constants. which brings a useful case of using Interface for declaring Constants. We have used both Class and interface for storing application wide constants and advantage of using Interface was that you can implement interface and can directly access constants without referring them with class name which was the case earlier when Class is used for storing Constants. Though after introduction of [static imports in Java 5](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html) this approach doesn't offer any benefit over Class approach.

3) All methods declared inside Java Interfaces are implicitly public and abstract, even if you don't use public or abstract keyword. you can not define any concrete method in interface. That's why interface is used to define contracts in terms of [variables](http://javarevisited.blogspot.com/2012/02/difference-between-instance-class-and.html) and methods and you can rely on its implementation for performing job.

4) In Java its legal for an interface to extend multiple interface. for example following code will run without any compilation error:

**interface** Session extends Serializable, Clonnable{ }

Here Session interface in Java is also a Serializable and Clonnable. This is not true for [Class in Java](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html) and one Class can only extend at most another Class. In Java one Class can implement multiple interfaces. They are required to provide implementation of all methods declared inside interface or they can declare themselves as [abstract class](http://javarevisited.blogspot.sg/2010/10/abstraction-in-java.html).

## Example of interface in Java

[Java interface example, java interface tutorial](http://javarevisited.blogspot.com/2012/01/get-set-default-character-encoding.html)Java standard library itself has many inbuilt interfaces like [Serializable](http://javarevisited.blogspot.sg/2011/04/top-10-java-serialization-interview.html), Clonnable, [Runnable](http://www.blogger.com/goog_1507016574)or Callable interface in Java.  ***Declaring interface is easy but making it correct in first attempt is hard*** but if you are in business of designing API then you need to get it right in first attempt because its not possible to modify interface once it released without breaking all its implementation. here is an example of declaring interface in Java :

**interface** SessionIDCreator **extends** **Serializable**, **Cloneable**{  
        [**String**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html) TYPE = "AUTOMATIC";  
        **int** createSessionId();  
    }  
    
    **class** SerialSessionIDCreator **implements** SessionIDCreator{  
  
        **private** **int** lastSessionId;

 @**Override**  
        **public** **int** createSessionId() {  
            **return** lastSessionId++;  
        }  
        
    }

In above *example of interface in Java*, SessionIDCreator is an interface while SerialSessionIDCreator is a implementation of interface. **@Override annotation can be used on interface method from Java 6 onwards**, so always try to use it. Its one of those coding practice which should be in your [code review checklist](http://javarevisited.blogspot.com/2011/09/code-review-checklist-best-practice.html).

## When to use interface in Java

*Interface* is best choice for **Type declaration** or defining contract between multiple parties. If multiple programmer are working in different module of project they still use each others API by defining interface and not waiting for actual implementation to be ready. This brings us lot of flexibility and speed in terms of coding and development. Use of Interface also ensures [best practices](http://javarevisited.blogspot.com/2011/08/code-comments-java-best-practices.html) like "**programming for interfaces than implementation**" and results in more flexible and maintainable code. Though interface in Java is not the only one who provides higher level abstraction, you can also use abstract class but choosing between Interface in Java and abstract class is a skill. Difference between Interface in Java and abstract class in java is also a very [popular java interview question](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html).

That's it for now on *What is Interface in Java,* specifics of Java interface and **How and when to use Interface in Java**.  Interface is key to write flexible and maintainable code. If you are not yet using interface in your code than start thinking in terms of interfaces and use it as much possible. You will learn more about interfaces when you start using design patterns. many design patterns like [decorator pattern](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html), [Factory method pattern](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html)  or [Observer design pattern](http://javarevisited.blogspot.com/2011/12/observer-design-pattern-java-example.html)  makes very good use of Java interfaces.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

<https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html>

# Default Methods

The section [Interfaces](https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html) 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](https://docs.oracle.com/javase/tutorial/java/IandI/examples/TimeClient.java), as described in [Answers to Questions and Exercises: Interfaces](https://docs.oracle.com/javase/tutorial/java/IandI/QandE/interfaces-answers.html):

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](https://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/SimpleTimeClient.java), implements TimeClient:

package defaultmethods;

import java.time.\*;

import java.lang.\*;

import java.util.\*;

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);

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](https://docs.oracle.com/javase/8/docs/api/java/time/ZonedDateTime.html) object (which is like a [LocalDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html) object except that it stores time zone information):

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](https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html) is a method declared without an implementation.)

package defaultmethods;

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();

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](https://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/TestSimpleTimeClient.java), invokes the method getZonedDateTime from an instance of SimpleTimeClient:

package defaultmethods;

import java.time.\*;

import java.lang.\*;

import java.util.\*;

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 nondefault (and nonstatic) 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](https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html) 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](https://docs.oracle.com/javase/8/docs/api/java/time/ZoneId.html) 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):

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.

## 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](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) interface has been enhanced with default and static methods.

Consider the Card and Deck classes as described in [Questions and Exercises: Classes](https://docs.oracle.com/javase/tutorial/java/javaOO/QandE/creating-questions.html). This example rewrites the [Card](https://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/Card.java) and [Deck](https://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/Deck.java) classes as interfaces. The Card interface contains two enum types (Suit and Rank) and two abstract methods (getSuit and getRank):

package defaultmethods;

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:

package defaultmethods;

import java.util.\*;

import java.util.stream.\*;

import java.lang.\*;

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](https://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/PlayingCard.java) implements the interface Card, and the class [StandardDeck](https://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/StandardDeck.java) implements the interface Deck.

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

public class StandardDeck implements Deck {

private List<Card> entireDeck;

// ...

public void sort() {

Collections.sort(entireDeck);

}

// ...

}

The method Collections.sort sorts an instance of List whose element type implements the interface [Comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html). 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](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html#compareTo-T-) method as follows:

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](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) interface to specify new sorting criteria, and use the method [sort(List<T> list, Comparator<? super T> c)](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#sort-java.util.List-java.util.Comparator-) (the version of the sort method that includes a Comparator parameter). You can define the following method in the class StandardDeck:

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](https://docs.oracle.com/javase/tutorial/java/IandI/examples/defaultmethods/SortByRankThenSuit.java) does this:

package defaultmethods;

import java.util.\*;

import java.util.stream.\*;

import java.lang.\*;

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 what you want to sort, not how you want to sort. 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](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach6), 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](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#comparing-java.util.function.Function-java.util.Comparator-):

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

In this example, you can use a [method reference](https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html) instead:

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

This invocation better demonstrates what to sort rather than how to do it.

The Comparator interface has been enhanced with other versions of the static method comparing such as [comparingDouble](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#comparingDouble-java.util.function.ToDoubleFunction-java.util.Comparator-) and [comparingLong](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#comparingLong-java.util.function.ToLongFunction-) 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](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#thenComparing-java.util.Comparator-):

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](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#thenComparingDouble-java.util.function.ToDoubleFunction-) and [thenComparingLong](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#thenComparingLong-java.util.function.ToLongFunction-)) 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](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html#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.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*