Interfaces

Reading:

- Java Tutorial: [http://docs.oracle.com/javase/tutorial/java/lang/createinterface.html](http://docs.oracle.com/javase/tutorial/java/lang/createinterface.html)
- Textbook: Chapter 13.1
Interfaces:

In the Java programming language, an interface is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types.

Method bodies exist only for default methods and static methods.

Interfaces cannot be instantiated—they can only be implemented by classes or extended by other interfaces.

An interface defines a protocol of behaviors that can be implemented by any class (anywhere in the class hierarchy).

Example situation 1:
An interface LinearDataStructure may have methods such as: search(), addElement() and removerElement().

Any linear data structure (e.g. Stack, Queue etc.) that implement LinearDataStructure must provide a body for all the methods listed in LinearDataStructure.
public interface LinearDataStructure{
    public void search ();
    public void addElement ();
    public void removeElement ();
}

//Implementing an Interface
class Stack implements LinearDataStructure{
    public void search (){        
        System.out.println("It is Stack. search") ;
    }
    public void addElement (){        
        System.out.println("It is Stack. addElement") ;
    }
    public void removeElement (){        
        System.out.println("It is Stack. removeElement") ;
    }
}   

class Queue implements LinearDataStructure{
    public void search (){        
        System.out.println("It is Queue. search") ;
    }
    public void addElement (){        
        System.out.println("It is Queue. addElement") ;
    }
    public void removeElement (){        
        System.out.println("It is Queue. removeElements") ;
    }
}   

}//end class Stack

}//end class Queue
//Using an Interface as a Type
public class Testing{
    public static void main (String [] args){
        Testing tst = new Testing();
        tst.aMethod(new Stack());
        tst.aMethod(new Queue());
    }
    public void aMethod (LinearDataStructure lds){
        lds.addElement();
        lds.removeElement();
        lds.search();
    }
} //end class testing
Example situation 2:

Imagine a futuristic society where computer-controlled robotic cars transport passengers through city streets without a human operator.

Automobile manufacturers write software (Java, of course) that operates the automobile—stop, start, accelerate, turn left, and so forth.

Another industrial group, electronic guidance instrument manufacturers, make computer systems that receive GPS (Global Positioning System) position data and wireless transmission of traffic conditions and use that information to drive the car.

The auto manufacturers must publish an industry-standard interface that spells out in detail what methods can be invoked to make the car move (any car, from any manufacturer).

The guidance manufacturers can then write software that invokes the methods described in the interface to command the car.

Neither industrial group needs to know how the other group's software is implemented. In fact, each group considers its software highly proprietary and reserves the right to modify it at any time, as long as it continues to adhere to the published interface.
public interface OperateCar {

    // constant declarations, if any

    // method signatures

    // An enum with values RIGHT, LEFT
    int turn(Direction direction,
             double radius,
             double startSpeed,
             double endSpeed);
    int changeLanes(Direction direction,
                     double startSpeed,
                     double endSpeed);
    int signalTurn(Direction direction,
                    boolean signalOn);
    int getRadarFront(double distanceToCar,
                       double speedOfCar);
    int getRadarRear(double distanceToCar,
                      double speedOfCar);

    ......

    // more method signatures
}

Note that the method signatures have no braces and are terminated with a semicolon.

To use an interface, you write a class that implements the interface. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface.
public class OperateBMW760i implements OperateCar {

    // the OperateCar method signatures, with implementation--
    // for example:
    int signalTurn(Direction direction, boolean signalOn) {
        // code to turn BMW's LEFT turn indicator lights on
        // code to turn BMW's LEFT turn indicator lights off
        // code to turn BMW's RIGHT turn indicator lights on
        // code to turn BMW's RIGHT turn indicator lights off
    }

    // other members, as needed -- for example, helper classes not
    // visible to clients of the interface
}

Interfaces as APIs
Interfaces are very useful for APIs.

Keep an interface public (to customers) and you may revise the implementation at a later date as long as you continue to implement the original interface that its customers have relied on.

(See the example scenario given in the tutorial)
Defining an Interface

An interface declaration consists of modifiers, the keyword `interface`, the interface name, a comma-separated list of parent interfaces (if any), and the interface body. For example:

```java
public interface GroupedInterface extends Interface1, Interface2, Interface3 {

    // constant declarations

    // base of natural logarithms
    double E = 2.718282;

    // method signatures
    void doSomething (int i, double x);
    int doSomethingElse(String s);
}
```

The `public` access specifier indicates that the interface can be used by any class in any package.

An interface can extend other interfaces, just as a class subclass or extend another class.

However, whereas a class can extend only one other class, an interface can extend any number of interfaces.
The Interface Body

The interface body can contain abstract methods, default methods, and static methods.

An abstract method within an interface is followed by a semicolon, but no braces (an abstract method does not contain an implementation).

Default methods are defined with the default modifier, and static methods with the static keyword.

All abstract, default, and static methods in an interface are implicitly public, so you can omit the public modifier.

In addition, an interface can contain constant declarations. All constant values defined in an interface are implicitly public, static, and final. Once again, you can omit these modifiers.

Implementing an Interface

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.
A Sample Interface, Relatable

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

```java
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.

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

```java
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, shown in bold in the previous example, 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`. 
Using an Interface as a Type

When you define a new interface, you are defining a new reference data type.

You can use interface names anywhere you can use any other data type name.

If you define a reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface.

```java
public Object findLargest(Object object1, Object object2) {
    Relatable obj1 = (Relatable)object1;
    Relatable obj2 = (Relatable)object2;
    if ((obj1).isLargerThan(obj2) > 0)
        return object1;
    else
        return object2;
}
```

By casting `object1` to a `Relatable` type, it can invoke the `isLargerThan` method.
Evolving Interfaces

Consider an interface that you have developed called DoIt:

```java
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:

```java
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

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:
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` 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`. (An abstract method is a method declared without an implementation.)

```java
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) {
        //code omitted
    }
}
```
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
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.

Not mention the default method at all

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

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

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

Redeclare the default method

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

```java
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.
Redefine the default method

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

```java
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.)

```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));
    }
}
```