Abstract Classes

Some classes in Java are purely artificial. Open the code example Polymorphism.java and review the Point class. Was an object of the type Point ever instantiated? No, we never created an object of this type. It was necessary for us to define the class Point, however we did not instantiate it. This class exists only as a common superclass for the others. Such classes are called abstract classes.

The syntax of abstract classes is the following: abstract classes declare functionality which all derived classes must implement, however, abstract classes themselves don't provide implementation, though they could have methods which are not abstract.

You cannot instantiate an abstract class. Nevertheless, abstract classes can have members and constructors. Constructors will be called when a derived class invoke super. Here is the Point class written as an abstract class

```java
public abstract class Point
{
    private double x, y;

    public Point(double a, double b) {x = a; y = b;}
    public double getX() {return x;}
    public double getY() {return y;}

    public abstract double area();

    public String toString()
        {return "(" + x + ", " + y + ")";}
}
```

In this class we have abstract as well as non-abstract methods. Having just one abstract method makes the whole class abstract. An abstract method is a dummy method that has no body. A subclass (or sublcasses) should provide an implementation for that method.

It is illegal to create an instance of an abstract class.

Abstract classes are helpful when some of the behavior is defined for most or all objects of a given type, but some behavior makes sense only for particular classes. In the above example, the method area does not make much sense for a point, however it does for a circle.

So as you can see, existence of one abstract method makes the whole class abstract. You cannot create an object out of an abstract class, however, you can use it to reference derived classes. An example,

```
Point x = new Point()    // illegal
Point y;
y = new Circle(1,2,3);    // legal (polymorphism)
```
Here are the rules and characteristics about abstract classes:

1. Any class with an abstract method is automatically abstract itself, and must be declared as such;
2. A class may be declared abstract even if it has no abstract methods. This prevents instantiation;
3. An abstract class cannot be instantiated;
4. A subclass of an abstract class can be instantiated if it implements each of the abstract methods;
5. If a subclass of an abstract class does not implement all of the abstract methods it inherits, that subclass is itself abstract.

**Interfaces**

After understanding abstract classes interfaces are a bit easier to understand. Interfaces are not classes; they are a separate entity. An interface looks like an abstract class except it uses the keyword `interface` instead of the words abstract and class. Interfaces are unique in the fact that they provide NO implementation! They provide a list of abstract methods which must be implemented by any class that implements that interface.

A class is said to **implement** the interface if it provides definitions for all abstract methods in the interface. You know that multiple inheritance is not allowed in java - a class can be derived only from one class. But multiple interfaces are allowed - a class may implement more than one interface.

If a parent class implements an interface, its child classes automatically implement the interface.

An interface, like a class, defines a type. Fields, variables, and parameters can be declared to be of a type defined by an interface.

Examples of API interfaces:

- Cloneable, Comparable, Serializable, ActionListener

The interface syntax is simple. Here is an example of `Comparable` interface which is defined in the `java.lang` package.

```java
public interface Comparable
{
    int compareTo(Object obj);
}
```

Methods declared in an interface are implicitly abstract and public (can have no other access modifier). By convention, the abstract modifier is omitted.
The idea of Comparable interface is to introduce an order in your set of objects. The advantage; once you implement this interface you can automatically sort your set of objects by using standard API methods.

Let us introduce an order in our set of geometrical shapes. Download the code example Abstract.java with the following classes: Point, Circle, Cylinder and Cone. We don't need to implement the Comparable interface in each of these classes. If we implement Comparable in one class, then by using inheritance we extend this to all other classes. It follows then, that we need only to implement Comparable in the abstract class Point.

```java
public abstract class Point implements Comparable
```

The keyword `implements` tells that the `Point` class has an implementation of `Comparable`. The keyword `abstract` here is just because `Point` was already abstract - not only abstract classes can implement interfaces, but any classes. `Comparable` is an extension of `==` sign. We say that two objects are equal if they have equal areas.

```java
public int compareTo(Object rhs)
{
    double diff = area() - ((Point) rhs).area();
    return (int) diff;
}
```

You can choose any other ordering, like for example, two solids are equal if their volumes are equal. So, we have two objects `x` and `y`, which can represent either Circle, or Cylinder or Cone. We will compare them as it follows

```java
Point x, y;
if (x.compareTo(y)) == 0)
    System.out.println("objects are identical");
```

The documentation of `compareTo` tells us that we have to implement it in such a fashion that it will be possible to check if `x.compareTo(y)) > 0` and `x.compareTo(y)) < 0`.

Once we implemented ordering, we can sort the array. This will print the array in ascending order:

```java
Arrays.sort(fig);
for(int i = 0; i < fig.length; i++)
    System.out.println(fig[i].getClass());
```

`getClass()` returns the runtime class of an object.

Because an interface is simply a list of unimplemented, and therefore abstract, methods, you might wonder how an interface differs from an abstract class. The differences are the following:
- An interface cannot implement any methods, whereas an abstract class can.
- A class can implement many interfaces but can have only one superclass.
- An interface is not part of the class hierarchy. Unrelated classes can implement the same interface.

**Static fields:**

An interface can declare three kinds of members:

- fields (constants)
- methods
- another interfaces (or nested classes).

All interface members are implicitly public, having non-public members would make little sense. Because interfaces contain no implementation, they cannot define normal fields. However, interfaces can define constants. Fields can be declared in an interface as either public static or public final. If no access level is given, its default is public. If a field is not explicitly declared to be static or final, the field is static final. These fields must be initialized. An example,

```java
public interface CheckingAccount
{
    int REGULAR = 0;    // static final is a default
    int GOLD = 1;

    void setCheckingAccount(some parameters);
}
```

**Extending Interfaces:**

Similar to classes interfaces can be extended using the keyword `extends`. Unlike classes, an interface can extend more than one interface. See the code example `Cruise.java`.

**Using Interfaces:**

In many practical cases the `Object` class does not provide enough genericity. Consider a simple example of finding the maximum in an array of Objects. Our generic code will only work if the method of finding the maximum is available in the `Object`. But that is not a case. In the previous lecture we considered an array of shapes, which did not have an order, a priori. To determine the order, we have to use the interface `Comparable` and implemented the `compareTo` method. Here is a code of the `findMax` function.

```java
public static Comparable findMax(Comparable[] A) //note the type
{
    int maxIndex = 0;
    for(int i = 0; i < A.length; i++)
        if ( A[i].compareTo(A[maxIndex]) > 0 )
            maxIndex = i;
```
return A[maxIndex];
}

Seems like interfaces is a way to go. Unfortunately, interfaces don't provide a complete solution as well. The main problem is that the above code works only for object that implement the `Comparable` interface, and thus the `compareTo` method. However, it is not always possible to decide on a "correct" meaning for `compareTo`. This method can be based on many different ideas, such as area, volume, perimeter, and so on. Once you implemented `compareTo`, you don't have much flexibility. But what if you want to have different comparison alternatives? The solution to this problem is to pass the comparison function as a parameter. Such comparison function in Java is implemented as a `Comparator`. Consider a poker hand – to do a fast evaluation you want to sort your cards either by value or by suit. The way to implement this is to pass a comparison function into a sorting routine:

```java
public void sortBySuit()
{
    Arrays.sort(hand, new SuitSort());
}

public void sortByValue()
{
    Arrays.sort(hand, new ValueSort());
}
```

In Java, you cannot pass a method; you should wrap a class around it. This new object is called a `functor`, and the style of programming is called a functional programming. The `functor` is a simple class which usually contains no data, but only a single method. In Java the functor is implemented via an interface.

```java
public class SuitSort implements java.util.Comparator
{
    public int compare(Object x, Object y)
    {
        int a = ((Card) x).getSuit();
        int b = ((Card) y).getSuit();

        return a-b;
    }
}
```

You can create your own interface, or can use a standard one. We can design different comparison functions by simply declaring new classes, one class for each kind of functionality. An instance of this class (which implements the interface) is passed to algorithm, which in turn calls the method from the function object. A more detailed example is shown in `Functor.java`.

**Nested/Inner Classes:**
A **nested** class is a class declaration that is placed inside another class declaration - the outer class. You do this by adding a keyword `static`. Typically, the nested class is private, and thus inaccessible from outside of the outer class.

A nested class that is not static is called an **inner** class. There are three types of inner classes:

- **Member Classes**
- **Local Classes**
- **Anonymous Classes**

1. Compiling a class that contains inner classes results in a separate `.class` file for every class. For the inner class you will see `#` key in its `.class` name.
2. Inner classes with class names can be defined as public, protected or private.
3. To access the outer class’s `this` reference, use `OuterClassName.this`.
4. The outer class is responsible for creating objects of its inner classes. To create an object of another class’s inner class, first create an object of the outer class and assign it to a reference (call it `ref`). Then use a statement of the form `ref.new InnerClass()` to create an inner class object.

**See the code example** `Shipment.java`.

**Anonymous Inner Classes:**

An anonymous class is a local class with no name. Since an anonymous class has no name, you can not declare a variable or field to be of that type. For this reason, anonymous classes must either be a subclass of another class or implement an interface. The basic syntax is:

```
new ParentClassName( parameters ) { classDefinitionGoesHere };
```

Anonymous inner classes are frequently used with GUI event handling. An example,

```java
Demp tmp = new Demo()
{
    public Object do(Object obj)
    {
        return obj;
    }
}
```