Chapter 8 : Interfaces & Inner Classes

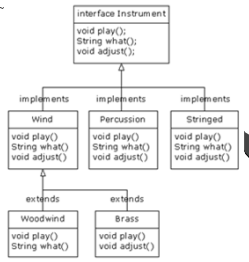
8.1 Interface

The **interface** keyword takes the **abstract** concept one step further. You could think of it as a "pure" **abstract** class.

It allows the creator to establish the form for a class, but no method bodies (no implementation).

An **interface** can also contain fields, but these are implicitly **static** and **final**.

To make a class conform to a particular **interface**, use the **implements** keyword.



Once you've implemented an **interface**, that implementation becomes an ordinary class that can be extended in the regular way.

When you **implement** an **interface**, the methods from the **interface** must be defined as **public**.

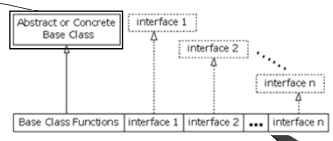
Sample : Music5.java

The rest of the code works the same. It doesn't matter if you are upcasting to a "regular" class called **Instrument**, an **abstract** class called **Instrument**, or to an **interface** called **Instrument**. The behavior is the same.

8.1.1 "Multiple inheritance" in Java

There is no storage associated with an **interface**, so there's nothing to prevent many **interface**s from being combined.

The problems seen in C++ do not occur with Java when combining multiple interfaces.



If you *do* inherit from a non-**interface**, you can inherit from only one. All the rest of the base elements must be **interface**s.

Sample : Adventure.java

When you combine a concrete class with interfaces this way, the concrete class must come first, then the interfaces.

You can inherit from an interface, but then you've got another **interface**. If you want to create an object of the new type, it must be a class with all definitions provided.

The core reason for interfaces is : to be able to upcast to more than one base type.

An **interface** gives you the benefits of an **abstract** class *and* an **interface**, so you should prefer **interface**s to **abstract** classes.

8.1.2 Extending an interface with inheritance

You can easily add new method declarations to an **interface** using inheritance, and you can also combine several **interface**s into a new **interface** with inheritance.

Sample : HorrorShow.java

Normally, you can use **extends** with only a single class, but **extends** can refer to multiple base interfaces when building a new **interface**.

8.1.3 Grouping constants

Sample : Months.java

Note : Java uses all uppercase letters (with underscores to separate multiple words in a single identifier) for **static final**s that have constant initializers.

The fields in an **interface** are automatically **public**.

Hint : Improvements in JDK 1.5

8.2 Inner classes

Placing a class definition within another class definition is called an *inner class.*

Inner class is a valuable feature because it allows you to group classes that logically belong together and to control the visibility of one within the other.

class

```
// "Constant Interface" antipattern - do not use!
public interface Physics {
  public static final double AVOGADROS_NUMBER    = 6.02214199e23;
  public static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
  public static final double ELECTRON_MASS       = 9.10938188e-31;
}

public class Guacamole implements Physics {
    public static void main(String[] args) {
        double moles = ....;
        double molecules = AVOGADROS_NUMBER * moles;
        ...
    }
}
```

This is a very bad idea: interfaces are for defining types, not providing constants.

```
import static org.iso.Physics.*;

class Guacamole {
    public static void main(String[] args) {
        double molecules = AVOGADROS_NUMBER * moles;
        ...
    }
}
```

Sample : Parcel1.java

More typically, an outer class will have a method that returns a reference to an inner class.

Sample : Parcel2.java

Note : If you want to make an object of the inner class anywhere except from within a non-**static** method of the outer class, you must specify the type of that object as *OuterClassName.InnerClassName.*

8.2.1 Inner classes and upcasting

Sample : Parcel3.java

In fact, you can't even downcast to a **private** inner class (or a **protected** inner class unless you're an inheritor), because you can't access the name.

The **private** inner class provides a way for the class designer to completely prevent any type-coding dependencies and to completely hide details about implementation.

Normal (non-inner) classes cannot be made **private** or **protected**.

8.2.2 Inner classes in methods and scopes

There are two reasons for doing this:

1. You're implementing an interface of some kind so that you can create and return a reference.
2. You want to create a class not publicly available.

❖ A class defined within a method :

Sample : Parcel4.java

PDestination cannot be accessed outside of **dest( )**.

❖ A class defined within a scope inside a method :

Sample : Parcel5.java

The class TrackingSlip isn't conditionally created—it gets compiled along with everything else.

❖ An anonymous class implementing an interface :

Sample : Parcel6.java

```
return new Contents() {
  private int i = 11;
  public int value() { return i; }
};
```

→

```
class MyContents implements Contents {
  private int i = 11;
  public int value() { return i; }
}
return new MyContents();
```

This syntax means: "Create an object of an anonymous class that's inherited from **Contents**." The reference returned by "**new**" is automatically upcast to a **Contents** reference.

❖ An anonymous class extending a class that has a nondefault constructor :

Sample : Parcel7.java

An anonymous class cannot have a constructor. You can, however, perform initialization at the point of definition of your fields.

❖ An anonymous class that performs field initialization :

Sample : Parcel8.java

Note :If you're defining an anonymous inner class and want to use an object that's defined outside the inner class, the compiler requires the outside object be **final**.

❖ An anonymous class that performs construction using instance initialization.

Sample : Parcel9.java

An instance initializer is the constructor for an anonymous inner class.

8.2.3 The link to the outer class

When you create an inner class, an object of that inner class has a link to the enclosing object that made it, and so it can access the members of that enclosing object—*without* any special qualifications.

Inner classes have access rights to all the elements in the enclosing class.

Sample : Sequence.java

8.2.4 static inner classes

- You don't need an outer-class object in order to create an object of a **static** inner class.
- You can't access an outer-class object from an object of a **static** inner class.
- **Static** inner classes can have **static** data, **static** fields, or **static** inner classes.

Sample : Parcel10.java

8.2.5 Referring to the outer class object

If you need to produce the reference to the outer class object, you name the outer class followed by a dot and **this**.

Sample : Parcel11.java

8.2.6 Inner class identifiers

Every class produces a **.class** file that holds all the information about how to create objects of this type. ---- a **Class** object

Inner classes also produce **.class** files to contain the information for *their* **Class** objects. ---- EnclosingClassName$InnerClassName.class

If inner classes are anonymous, the compiler simply starts generating numbers as inner class identifiers.

8.2.7 Why inner classes?

The inner class inherits from a class or implements an **interface**, and the code in the inner class manipulates the outer class object. ---- An inner class provides a kind of window into the outer class.

Each inner class can independently inherit from an implementation. One way to look at the inner class is as the completion of the solution of the multiple-inheritance problem.

Sample : MultiInterfaces.java ⟷ Sample : MultiImplementation.java

8.2.8 Inner classes & control frameworks

An *application framework* is a class or a set of classes that's designed to solve a particular type of problem. To apply an application framework, you inherit from one or more classes and override some of the methods.

The control framework is a particular type of application framework dominated by the need to respond to events.

The Java Swing library is a control framework that elegantly solves the GUI problem and that heavily uses inner classes.

Inner classes allow two things in a control framework :

1. To create the entire implementation of a control-framework application in a single class.
2. You're able to easily access any of the members in the outer class.

Sample : GreenhouseControls

Inner classes allow you to have multiple derived versions of the same base class within a single class.