Contents

Packages

Packages provide for us a way to name-space and compartmentalize a set of classes for some specific purpose. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are exposed only to other members of the same package.

To create a package is quite easy: simply include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name.

Java uses file system directories to store packages Remember that case is significant, and the directory name must match the package name exactly. More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multi leveled package statement is shown here: `package pkg1[.pkg2[.pkg3]];` A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as package `java.awt.image;` needs to be stored in `java\awt\image` in a Windows environment.

`First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable. Third, you can use the -classpath option with java and javac to specify the path to your classes.`

Access

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code.

`A non-nested class has only two possible access levels: default and public. When a class is declared as public, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.`

```
package pk1;
```

```java
public class ClassOne {
    int defaultAccess;
    public publicAccess;
    private privateAccess;
    protected protectedAccess;
}

package pk1;

public class ClassTwo {

    public void method() {
        ClassOne one = new ClassOne();
        one.defaultAccess; // accessible since we are in the same package
           pk1
        one.publicAccess; // accessible from everywhere since it's public
        one.protectedAccess; // compile time error due to protected access
        one.privateAccess; // compile time error due to private access
    }
}

package pk2;

public class ClassThree {

    public void method() {
        ClassOne one = new ClassOne();
        one.publicAccess; // accessible from everywhere since it's public
        one.defaultAccess; // compile time error not in the same package
        one.protectedAccess; // compile time error due to protected access
        one.privateAccess; // compile time error due to private access
    }
}

package pk3;

public class ClassFour extends ClassTwo {

    public void method() {
        this.publicAccess; // accessible from everywhere since it's public
        this.protectedAccess; // accessible protected access but from
           inheritance
        this.defaultAccess; // compile time error we are in a different
           package
        this.privateAccess; // compile time error due to private access
    }
}
```

Importing

Java includes the import statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The import statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the import statement will save a lot of typing. In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions. This is the general form of the import statement: `import pkg1[.pkg2].(classname | *);`

It must be emphasized that the import statement is optional. Any place you use a class name, you can use its fully qualified name, which includes its full package hierarchy

```
import java.util.Date;
import java.io.*;
```

`All of the standard Java classes included with Java are stored in a package called java. The basic language functions are stored in a package inside of the java package called java.lang. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in java.lang, it is implicitly imported by the compiler for all programs.`

If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

Interfaces

Using the keyword interface, you can fully abstract a class' interface from its implementation. That is, using interface, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body

Beginning with JDK 8, it is possible to add a default implementation to an interface method. Thus, it is now possible for interface to specify some behavior. However, default methods constitute what is, in essence, a special-use feature, and the original intent behind interface still remains.

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition

Visibility

By default all members of an interface are defined as `public` and so is the interface itself, it has little to no value to have interface types default to the default package-private access visibility as usual classes and their members do when the access modifier is omitted.

```
interface PublicAccess { // the interface is implicitly public
    void method(); // the method is implicitly public
}
```

Constants

One can use an interface to package or bundle a set of constants which are to be used in other classes which implement the interface, by default these constants defined in an interface are implicitly `public static final` even though we do not need to specify it, and it can not be any other way since interfaces have no instance member variables. Java also gives the possiblity to define static methods in an interface, however they have some caveats (see below) compared to static members defined in normal classes

```
interface SharedConstants {
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;

    static int compute() {
        return 0;
    }
}
class ImplementingConstants {

    ImplementingConstants() {
        int f = NO; // this is valid we need not prefix the constant usage
           with the name of the interface
    }
}
```

It is generally not used, and often frowned upon, since constants can be defined in normal classes exposed explicitly as `public static final` instead. To access a static constant from an interface which our class implements, we only need to reference its identifier, as shown above, unlike static methods (see below) which can not be accessed like that.

Static methods from interfaces are not inherited, meaning that we can not call the static method from the interface with `this.method` or `method`, they have to be explicitly prefixed with the name of the interface `Interface.method`

```
interface B {
    static void method() {
        return;
    }
}
public static class K {
    static void another() {}
}
class C extends K implements A, B {

    public C() {
        method(); // invalid, this would produce a compile time error
        B.method(); // this is valid, prefixed with the interface name

        this.another(); // this is valid, inherited from regular class K
        another(); // this is valid, inherited from regular class K
    }
}
```

Implementing

```
class Classname [extends superclass] [implements interface
   [,interface...]] {
```

```
// class-body
}
```

Note that unlike the relationship between child classes and base classes in an inheritance, a given class can implement any number interfaces. This is because they specify behavior, not state, and the usual drawbacks of multiple inheritance are completely avoided

The diamond problem (also called the diamond inheritance problem) arises in object-oriented programming languages that allow multiple inheritance, where a class can inherit from more than one parent class. The issue occurs when a class inherits from two classes that both inherit from the same superclass, creating an inheritance structure shaped like a diamond.

```
   A
  / \
 B   C
  \ /
   D
```

`test`

In this example, class D inherits from both B and C, and both B and C inherit from A. As a result, D inherits two copies of A, and there is ambiguity when calling or using class members

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.

Extending

An interface itself can be extended to `inherit` behavior from other interfaces, unlike with class inheritance here interfaces can extend from multiple other interfaces, which in effect extends their behavior in a reusable and optimal way

```
interface A {
    void methodA();
}
interface B {
    void methodB();
}
interface C extends A, B {
    void methodC();
}
class D implements C {
    // must implement all 3 methods from A,B and C
}
```

Defaults

The release of JDK 8 has changed this by adding a new capability to interface called the default method. A default method lets you define a default implementation for an interface method.

`A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code. Recall that there must be implementations for all methods defined by an interface.`

```
interface Sequence {
    default void remove() { // added later on in the implementation cycle
    }
    default void insert() { // added later on in the implementation cycle
    }
    void inspect(); // added with the initial implementation cycle
}
class Immutable implements Sequence {
    void inspect() {
    }
}
class Mutable implements Sequence {
    void remove() {
    }
    void insert() {
    }
    void inspect() {
    }
}
```

In this example we can see how the methods which were possibly added later could interfere with the `Immutable` class since we would now have to go in and add stub empty implementations for these methods in that class, with default methods we can `extend` the behavior of a given interface without worrying about breaking existing implementations as long as the extending of the behavior makes sense, otherwise a new interface should be created instead.

Caveats

As mentioned multiple inheritance is not allowed in java, however we also saw that a class can implement multiple interfaces, now it was also noted that each interface can have a default method, what would happen if a clsss implements from two interfaces both of which provide a default method with the same signature. If two interfaces provide the same signature but it is not default implemented method, since interfaces have no instance, the signature of one interface will shadow the other, but since they are the same that will not produce any compile time error.

```
    interface A {
        default void methodOne() {}
        default void methodTwo() {}
    }

    interface B {
        default void methodOne() {}
        default void methodTwo() {}
    }

    class C implements A, B {

        public void methodOne() {} // this is valid, the local one
           overrules the default implementations from A and B
        // compile time error methodTwo has not been overridden in the
           current class, thus no way to resolve methodTwo
    }
```