Packages and Interfaces

Ebey S.Raj

Packages

Introduction

- The name of the class was taken from the same namespace.
  - A unique name had to be used for each class to avoid name collisions.
- Java provides a mechanism called **Packages** for partitioning the class namespace into more manageable chunks.

Introduction

- **Packages** are containers for classes.
  - Eg: a package allows you to create a class named **Shape**, which you can store in your own package without concern that it will collide with some other class named **Shape** stored elsewhere.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

Packages

- Package is both a naming and a visibility control mechanism.
- Classes inside a package
  - are exposed only to other members of the same package
  - are not accessible by code outside that package.

Defining a package

- To create a package, simply include a package command as the first statement in a Java source file.
- 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.
- This is the general form of the package statement:

package *pkg*;

Here, pkg is the name of the package.

- Eg: Following statement creates a package called MyPackage:

package *MyPackage*;

Defining a package

- Java uses file system **directories** to store packages.
  - For example, the .class files for any classes you declare to be part of MyPackage must be stored in a directory called MyPackage.
- The directory name must **match** the package name exactly.
- More than one file can include the same package statement.

Defining a package

- We 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 multileveled package statement is shown here:

package *pkg1[.pkg2[.pkg3]]*;

- A package hierarchy must be reflected in the file system of your Java development system.

You cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and classpath

- How does the Java run-time system know where to look for packages that you create?
  - 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.
  - Specify a directory path or paths by setting the CLASSPATH environmental variable.
  - Use the -classpath option with java and javac to specify the path to your classes.

Finding Packages and classpath

- When the second two options are used, the class path must not include MyPack, itself.
  - It must simply specify the path to MyPack. For example, in a Windows environment, if the path to MyPack is "C:\MyPrograms\Java\MyPack". Then the class path to MyPack is

    C:\MyPrograms\Java

Access Protection

- 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. The class is Java's smallest unit of abstraction.

Access Protection

- Java addresses four categories of visibility for class members:
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses

Access Protection

- The three access modifiers, private, public, and protected.
- Anything declared **public** can be accessed from anywhere.
- Anything declared **private** cannot be seen outside of its class.
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the **default** access.
- If you want to allow an element to be seen outside your current package, but only to immediate subclasses, then declare that element as **protected**.

Access Protection

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Access Protection

- 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.

Access Protection

- Examples
  - P1.Protected.java
  - P1.Derived.java
  - P1.SamePackage.java
  - P2.Protection2.java (Derived from Protection)
  - P2.OtherPackage.java
  - P1.AccessDemo.java
  - P2.AccessDemo.java

Importing Packages

- Classes within packages must be fully qualified with their package name or names
  - Tedious to type in the long dot-separated package path name for every class you want to use.
- 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.

P2.Protection2.java (Derived from Protection)

Importing Packages

- In a Java source file, import statements occur immediately following the package statement and before any class definitions.
- General form of the **import** statement:

```
import pkg1[.pkg2].(classname|*);
```

Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.). Finally, you specify either an explicit classname or a star (*)[for entire package]

- There is no practical limit on the depth of a package hierarchy.

Importing Packages: Example

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

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**.

If a class with the same name exists in two different packages and we want to import both the classes, then we have to explicitly name the class specifying its package.

Interfaces

Interfaces

- Java allows us to fully abstract a class's interface from its implementation through the use of **Interface** keyword.
- 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 **their methods are declared without any body.**
- Any number of classes can implement an interface.
- One class can implement any number of interfaces.

Interfaces

- Although they are similar to abstract classes, interfaces have an additional capability:
  - A class can **implement more than one interface**. By contrast, **a class can only inherit a single superclass**.
- To implement an interface, a class **must create the complete set of methods** defined by the interface.
  - Each class is free to determine the details of its own implementation.
- By providing the interface keyword, Java allows us to fully utilize the **“one interface, multiple methods”** aspect of polymorphism.

Defining Interfaces

- An interface is defined much like a class. This is a simplified general form of an interface:

```
access interface name {
    return-type method-name1(parameter-list);
    return-type method-name2(parameter-list);
    type final-varname1 = value;
    type final-varname2 = value;
    //...
    return-type method-nameN(parameter-list);
    type final-varnameN = value;
}
```

Defining Interfaces

- No Access Modifier is included → default access
  - The interface is only available to other **members of the package** in which it is declared.
- Public Access Modifier is included
  - The interface can be used by **any other code**.
  - The interface must be the **only public interface declared in the file**, and the file must have **the same name** as the interface.
    - *name* is the name of the interface, and can be any valid identifier.

Defining Interfaces

- The methods that are declared in an interface are **abstract methods.**
- Variables can be declared inside of interface declarations.
  - They are implicitly **final and static**, meaning they cannot be changed by the implementing class.
  - They must also be **initialized**.
  - All methods and variables are **implicitly public**.

Interface Definition: Example

```
interface Callback {
        void callback(int param);
}
```

Implementing Interfaces

- 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, and then create the methods defined by the interface.
- General form

```
class classname [extends superclass] [implements interface [,interface...]] {
// class-body
}
```

Implementing Interfaces

- The methods that implement an interface must be declared **public.**
- The type signature of the implementing method must match exactly the type signature specified in the interface definition.

```
class Client implements Callback{
        // Implement Callback's interface
        public void callback(int p) {
                System.out.println("callback called with " + p);
        }
}
```

Notice that **callback( )** is declared using the **public** access modifier.

Implementing Interfaces

- Classes that implement interfaces can define additional members of their own.

```
class Client implements Callback{
        // Implement Callback's interface
        public void callback(int p) {
                System.out.println("callback called with " + p);
        }
        void nonIfaceMeth() {
        System.out.println("Classes can define other members,too.");
        }
}
```

Accessing Implementations Through Interface References

- We can declare object references variables of an interface.
  - Any instance of any class that implements the declared interface can be referred to by such a variable.
  - The correct version will be called based on the **actual instance of the interface being referred to**.
- This is similar to using a superclass reference to access a subclass object.

Interface Example: P1.TestIface.java

Partial Implementations

- If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as abstract.

```
abstract class Incomplete implements Callback{
        int a, b;
        void show() {
                System.out.println(a + " " + b);
        }
        //...
}
```

Nested Interfaces

- An interface can be declared a member of a class or another interface. Such an interface is called a member interface or a nested interface.
- A nested interface can be declared as public, private, or protected.
- When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member.

Nested Interface Example: NestedIFDemo.java

Variables in Interface

- We 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.
- When you implement that interface in a class, all of those variable names will be in scope as constants
- If an interface contains no methods, then any class that includes such an interface imports the constant fields into the class name space as **final variables**.

Variables in Interface Example: AskMe.java

Interfaces Can Be Extended

- One interface can inherit another by use of the keyword **extends**.
  - The syntax is the same as for inheriting classes.
- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Extending Interfaces Example: IFExtend.java

References

- Lafore R., Object Oriented Programming in C++, Galgotia Publications, 2001.
- Balagurusamy, Object Oriented Programming with C++, Tata McGraw Hill, 2008.

Thank You

Ebey S.Raj
Asst Professor in IT
Govt Engg College, Sreekrishnapuram