Packages And Interfaces

- Packages are containers for classes. They are used to keep the class name space compartmentalized.
- For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere.
- Packages are stored in a hierarchical manner and are explicitly imported into new class
  definitions
- Using interface, you can specify a set of methods that can be implemented by one or more classes.
- In its traditional form, the interface, itself, does not actually define any implementation.
- 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 (abstract or otherwise).

Packages

- You need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers.
- Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks.
- This mechanism is the package. 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.
- This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

Defining a 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.
- While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.
- This is the general form of the package statement:
  package pkg;
- Here, pkg is the name of the package. For example, the following statement creates a
  package called MyPackage.
- 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.
- Remember that case is significant, and the directory name must match the package name
  exactly.
- 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 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. For example, a package declared as
  package java.awt.image;
- needs to be stored in java\awt\image in a Windows environment. Be sure to choose your
  package names carefully. You cannot rename a package without renaming the directory in
  which the classes are stored.

Finding Packages and CLASSPATH

- Packages are mirrored by directories.
- 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.
- In order for a program to find the a package, one of three things mentioned above must be true.
- Either the program can be executed from a directory immediately above MyPack, or the CLASSPATH must be set to include the path to MyPack, or the -classpath option must specify the path to package when the program is run via java.

```java
package MyPack; class Balance { String name; double bal;
Balance(String n, double b) { name = n; bal = b; } void show() {
if(bal<0) System.out.print("-->> "); System.out.println(name + ": $"
+ bal); } } class AccountBalance { public static void main(String
args[]) { Balance current[] = new Balance[3]; current[0] = new
Balance("K. J. Fielding", 123.23); current[1] = new Balance("Will
Tell", 157.02); current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show(); } }
```

Java

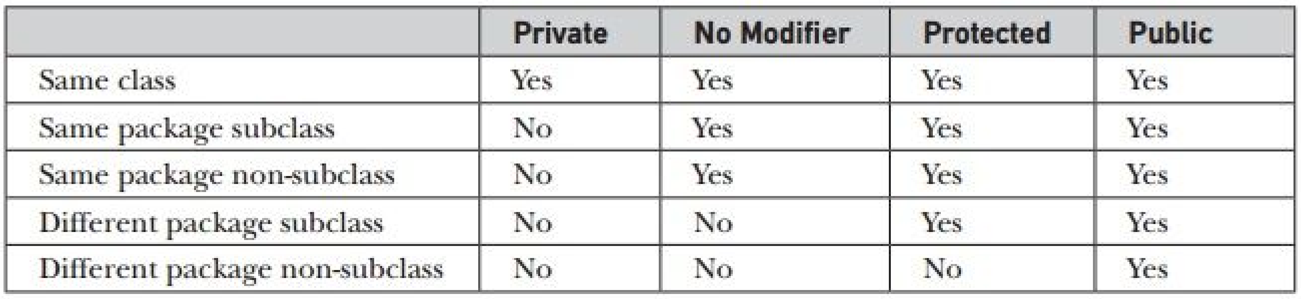
- Call this file AccountBalance.java and put it in a directory called MyPack.
- Next, compile the file. Make sure that the resulting .class file is also in the MyPack
  directory. Then, try executing the AccountBalance class, using the following command line:
  java MyPack.AccountBalance*;*

Access Protection

- Access to a private member of a class is granted only to other members of that class. Packages add another dimension to access control.

- Classes and packages are both means of encapsulating and containing the name space
  and scope of variables and methods.

- 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
- The three access modifiers, private, public, and protected, provide a variety of ways to
  produce the many levels of access required by these categories.
- While Java's access control mechanism may seem complicated, we can simplify it as
  follows. 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 classes that subclass your class directly, then declare that element ***protected***.



	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

Table 9-1 Class Member Access

An Access Example

- The following example shows all combinations of the access control modifiers. This
  example has two packages and five classes.
- Remember that the classes for the two different packages need to be stored in directories named after their respective packages—in thiscase, p1 and p2.

- The source for the first package defines three classes: Protection, Derived, and SamePackage.
- The first class defines four int variables in each of the legal protection modes.
- The variable n is declared with the default protection, n_pri is private, n_pro is protected, and n_pub is public.

```
package p2; class Protection2 extends p1.Protection { Protection2() {
System.out.println("derived other package constructor"); // class or
package only // System.out.println("n = " + n); // class only //
System.out.println("n_pri = " + n_pri); System.out.println("n_pro = "
+ n_pro); System.out.println("n_pub = " + n_pub); } } class
OtherPackage { OtherPackage() { p1.Protection p = new
p1.Protection(); System.out.println("other package constructor"); //
class or package only // System.out.println("n = " + p.n); // class
only // System.out.println("n_pri = " + p.n_pri); // class, subclass
or package only // System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub); } }
```

Java

Importing Packages

- Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use.
- For this reason, 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 | *);

- 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 (.). There is no practical limit on the
  depth of a package hierarchy, except that imposed by the file system.
- Finally, you specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use:
  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.

```
package MyPack; /* Now, the Balance class, its constructor, and its
show() method are public. This means that they can be used by non-
subclass code outside their package. */ public class Balance { String
name; double bal; public Balance(String n, double b) { name = n; bal
= b; } public void show() { if(bal<0) System.out.print("-->> ");
System.out.println(name + ": $" + bal); } }
```

Java

- As you can see, the Balance class is now public. Also, its constructor and its show( )
  method are public, too.
- This means that they can be accessed by any type of code outside the MyPack package.
- For example, here TestBalance imports MyPack and is then able to make use of the Balance class:

```
import MyPack.*; class TestBalance { public static void main(String
args[]) { /* Because Balance is public, you may use Balance class and
call its constructor. */ Balance test = new Balance("J. J. Jaspers",
99.88); test.show(); // you may also call show() } }
```

Java

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.
- In practice, this means that you can define interfaces that don't make assumptions about how they are implemented.
- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

Defining an Interface

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

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

Java

- When no access modifier is included, then default access results, and the interface is only
  available to other members of the package in which it is declared.
- When it is declared as public, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface.
- The methods that are declared have no bodies. They end with a semicolon after the parameter
  list.
- They are, essentially, abstract methods. Each class that includes such an interface must
  implement all of the methods.
- 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.

- Therefore, as a general rule, you will still often create and use interfaces in which no default methods exist.

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 required by the interface. The general form of a class that includes the
  implements clause looks like this:
  class classname [extends superclass] [implements interface [,interface...]] { /
- If a class implements more than one interface, the interfaces are separated with a comma.
  If a class implements two interfaces that declare the same method, then the same method
  will be used by clients of either interface.
- The methods that implement an interface must be declared public.
- Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.

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

Java

- Notice that callback( ) is declared using the public access modifier.REMEMBER When you implement an interface method, it must be declared as public.
- It is both permissible and common for classes that implement interfaces to define
  additional members of their own.
- For example, the following version of Client implements callback( ) and adds the method nonIfaceMeth( ):

```java
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 that
implement interfaces " + "may also define other members, too."); } }
```

Java

Accessing Implementations Through Interface References

- You can declare variables as object references that use an interface rather than a class type.
  Any instance of any class that implements the declared interface can be referred to by such
  a variable.
- When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces.
- The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them.
- The calling code can dispatch through an interface without having to know anything about the “callee.”. Here's an example

```java
class TestIface { public static void main(String args[]) { Callback c
= new Client(); c.callback(42); } }
```

Java

```java
Output : callback called with 42
```

Java

- Notice that variable c is declared to be of the interface type Callback, yet it was assigned an
  instance of Client.
- Although c can be used to access the callback( ) method, it cannot access any other members of the Client class.
- An interface reference variable has knowledge only of the methods declared by its interface declaration. Thus, c could not be used to access nonIfaceMeth( ) since it is defined by Client but not Callback.

- While the preceding example shows, mechanically, how an interface reference variable
  can access an implementation object, it does not demonstrate the polymorphic power of
  such a reference.

- To sample this usage, first create the second implementation of Callback, shown here:

```java
class AnotherClient implements Callback { // Implement Callback's
interface public void callback(int p) { System.out.println("Another
version of callback"); System.out.println("p squared is " + (p*p)); }
}
```
Java

```java
class TestIface2 { public static void main(String args[]) { Callback
c = new Client(); AnotherClient ob = new AnotherClient();
c.callback(42); c = ob; // c now refers to AnotherClient object
c.callback(42); } }
```
Java

```
Output : callback called with 42 Another version of callback p
squared is 1764
```
Java

- As you can see, the version of callback( ) that is called is determined by the type of object
  that c refers to at run time.

Partial Implementations

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

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

- Here, the class Incomplete does not implement callback( ) and must be declared as abstract.

- Any class that inherits Incomplete must implement callback( ) or be declared abstract itself.

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. This differs from a top-level interface, which must either be declared as public or use the default access level, as previously described.
- 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.
- Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified. Here is an example.

```java
class A { // this is a nested interface public interface NestedIF {
boolean isNotNegative(int x); } } // B implements the nested
interface. class B implements A.NestedIF { public boolean
isNotNegative(int x) { return x < 0 ? false : true; } } class
NestedIFDemo { public static void main(String args[]) { // use a
nested interface reference A.NestedIF nif = new B();
if(nif.isNotNegative(10)) System.out.println("10 is not negative");
if(nif.isNotNegative(-12)) System.out.println("this won't be
displayed"); } }
```

Java

- Notice that the name is fully qualified by the enclosing class' name. Inside the main( )
  method, an A.NestedIF reference called nif is created, and it is assigned a reference to
  a B object. Because B implements A.NestedIF, this is legal.

Variables in Interfaces

- 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.

- When you include that interface in a class (that is, when you “implement” the interface), all of those variable names will be in scope as constants. (This is similar to using a header file in C/C++ to create a large number of #defined constants or const declarations.)

- If an interface contains no methods, then any class that includes such an interface doesn’t actually implement anything.
- It is as if that class were importing the constant fields into the class name space as final variables.
- The next example uses this technique to implement an automated “decision maker”:

```java
import java.util.Random; interface SharedConstants { int NO = 0; int
YES = 1; int MAYBE = 2; int LATER = 3; int SOON = 4; int NEVER = 5; }
class Question implements SharedConstants { Random rand = new
Random(); int ask() { int prob = (int) (100 * rand.nextDouble()); if
(prob < 30) return NO; // 30% else if (prob < 60) return YES; // 30%
else if (prob < 75) return LATER; // 15% else if (prob < 98) return
SOON; // 13% else return NEVER; // 2% } } class AskMe implements
SharedConstants { static void answer(int result) { switch(result) {
case NO: System.out.println("No"); break; case YES:
System.out.println("Yes"); break; case MAYBE:
System.out.println("Maybe"); break; case LATER:
System.out.println("Later"); break; case SOON:
System.out.println("Soon"); break; case NEVER:
System.out.println("Never"); break; } } public static void
main(String args[]) { Question q = new Question(); answer(q.ask());
answer(q.ask()); answer(q.ask()); answer(q.ask()); } }
```

Java

```
Output : Later Soon No Yes
```

Java

- Notice that this program makes use of one of Java’s standard classes: Random. This class
  provides pseudorandom numbers. It contains several methods that allow you to obtain
  random numbers in the form required by your program.
- In this example, the method ***nextDouble( )*** is used. It returns random numbers in the range 0.0 to 1.0.

- In this sample program, the two classes, Question and AskMe, both implement the
  SharedConstants interface where NO, YES, MAYBE, SOON, LATER, and NEVER are defined.
- Inside each class, the code refers to these constants as if each class had defined or
  inherited them directly.

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 required by the interface inheritance chain.

```java
interface A { void meth1(); void meth2(); } // B now includes meth1() and meth2() -- it adds meth3(). interface B extends A { void meth3(); } // This class must implement all of A and B class MyClass implements B { public void meth1() { System.out.println("Implement meth1()."); } public void meth2() { System.out.println("Implement meth2()."); } public void meth3() { System.out.println("Implement meth3()."); } } class IFExtend { public static void main(String arg[]) { MyClass ob = new MyClass(); ob.meth1(); ob.meth2(); ob.meth3(); } }
```

Java

Default Interface Methods

- Prior to JDK 8, an interface could not define any implementation whatsoever. This meant that for all previous versions of Java, the methods specified by an interface were abstract, containing no body. This is the traditional form of an interface.
- 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. In other words, by use of a default method, it is now possible for an interface method to provide a body, rather than being abstract.

- During its development, the default method was also referred to as an extension method, and you will likely see both terms used.

- 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.

Default Method Fundamentals

- An interface default method is defined similar to the way a method is defined by a class.
- The primary difference is that the declaration is preceded by the keyword default. For
example, consider this simple interface: Consider the example

```java
public interface MyIF { // This is a "normal" interface method
declaration. // It does NOT define a default implementation. int
getNumber(); // This is a default method. Notice that it provides a
default implementation. default String getString() { return "Default
String"; } } // Implement MyIF. class MyIFImp implements MyIF { //
Only getNumber() defined by MyIF needs to be implemented. //
getString() can be allowed to default. public int getNumber() {
return 100; } } class DefaultMethodDemo { public static void
main(String args[]) { MyIFImp obj = new MyIFImp(); // Can call
getNumber(), because it is explicitly // implemented by MyIFImp:
System.out.println(obj.getNumber()); // Can also call getString(),
because of default // implementation:
System.out.println(obj.getString()); } }
```

Java

- MyIF declares two methods. The first, getNumber( ), is a standard interface method
declaration. It defines no implementation whatsoever. The second method is getString( ),
and it does include a default implementation.
- In this case, it simply returns the string "Default String". Pay special attention to the way getString( ) is declared. Its declaration is preceded by the default modifier.
- This syntax can be generalized. To define a default method, precede its declaration with **default.**

Multiple Inheritance Issues

- Java does not support the multiple inheritance of classes. Now that an interface can include default methods, you might be wondering if an interface can provide a way around this restriction.
- The answer is, essentially, no. Recall that there is still a key difference between a class and an interface: a class can maintain state information (especially through the use of instance variables), but an interface cannot
- For xample, assume that two interfaces called Alpha and Beta are implemented by a
  class called MyClass. What happens if both Alpha and Beta provide a method called reset( )
  for which both declare a default implementation? Is the version by Alpha or the version by
  Beta used by MyClass?
- Or, consider a situation in which Beta extends Alpha. Which version
  of the default method is used? Or, what if MyClass provides its own implementation of themethod? To handle these and other similar types of situations, Java defines a set of rules
  that resolves such conflicts.
- First, in all cases, a class implementation takes priority over an interface default implementation. Thus, if MyClass provides an override of the reset( ) default method,
  MyClass' version is used. This is the case even if MyClass implements both Alpha and Beta.
- In this case, both defaults are overridden by MyClass' implementation. Second, in cases in which a class implements two interfaces that both have the same default method, but the class does not override that method, then an error will result.
- Continuing with the example, if MyClass implements both Alpha and Beta, but does not
  override reset( ), then an error will occur.

Use static Methods in an Interface

- JDK 8 added another new capability to interface: the ability to define one or more static methods. Like static methods in a class, a static method defined by an interface can be
  called independently of any object.

- Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a static method.
- Instead, a static method is called by specifying the interface name, followed by a period, followed by the method name. Here is the general form: ***InterfaceName.staticMethodName***

```java
public interface MyIF { // This is a "normal" interface method declaration. // It does NOT define a default implementation. int getNumber(); // This is a default method. Notice that it provides // a default implementation. default String getString() { return "Default String"; } // This is a static interface method. static int getDefaultNumber() { return 0; } }
```

Java

- Virtually all real programs that you write in Java will be contained within packages. A number of them will probably implement interfaces as well.