Packages

- *Packages are containers for classes that are used to keep the class name* space compartmentalized.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.
- Defining a Package

```
    package MyPackage;
```

- 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 our Java development system. For example, a package declared as

  package java.awt.image;

  needs to be stored in **java/awt/image**

- packages are mirrored by directories. So how Java run-time system know where to look for packages that we create?
- The answer has two parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in the current directory, or 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.

```
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(“Ajay", 123.23);
current[1] = new Balance(“vijay", 157.02);
current[2] = new Balance(“sanjay", 12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

- 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
  ```
- Remember, we will need to be in the directory above MyPack when you execute this command, or to have your CLASSPATH environmental variable set appropriately.
- AccountBalance is now part of the package MyPack. So it cannot be executed by itself. That is, you cannot use this command line: `java AccountBalance`
- AccountBalance must be qualified with its package name.

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
- The three access specifiers, **private, public, and protected, provide a variety of** ways to produce the any levels of access required by these categories.

Importing Packages

- 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|*);*

  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. it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs:

  import java.lang.*;

- Any place we use a class name, we can use its fully qualified name, which includes its full package hierarchy. For example, this fragment uses an import statement:

```
import java.util.*;
 class MyDate extends Date {
}
```

- The same example without the import statement looks like this:

```
class MyDate extends java.util.Date {
}
```

- when a package is imported, only those items within the package declared as public will be available to non-subclasses in the importing code.

```
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);
}
}
```

```
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("Jayak", 99.88);
test.show(); // you may also call show()
}
}
```

- 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

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 their methods are declared without any body.
- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.
- By providing the interface keyword, Java allows us to fully utilize the "one interface, multiple methods" aspect of polymorphism.
- *Interfaces add most of the functionality that is required for many applications which would normally resort to using multiple inheritance in a language such as C++.*

Defining an Interface

An interface is defined much like a class. This is the 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;
}
```

- Here, *access is either public or not used. When no access specifier 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.
- The methods are abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the 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 with a constant value. All methods and variables are implicitly public if the interface, itself, is declared as public.

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. The general form of a class that includes the implements clause looks like this:

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

- Here, *access is either public or not used. 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.

```
interface Callback {
void callback(int param);
}
class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("callback called with " + p);
}
void disp() {
System.out.println("Classes that implement interfaces " +
"may also define other members, too.");
}
}
```

** When you implement an interface method, it must be declared as **public.***

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

- The 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 only has knowledge of the methods declared by its interface declaration.
- Thus, c could not be used to access disp( ) since it is defined by Client but not Callback.

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);
}
// ...
}
```

Interfaces Can Be Extended

- One interface can inherit another by use of the keyword **extends. The syntax is the** same as for inheriting classes.

```
// One interface can extend another.
interface A {
void meth1();
void meth2();
}
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
void meth3();
}
```