Packages & Interfaces

Package ---
A java package is a group of similar types of classes, interfaces and sub-packages.

Introduction

- The name of each example class so far was taken from the same namespace.
- That is why, a unique name had to be used for each class to avoid name collisions with class names chosen by other programmers
- Thankfully, Java provides a mechanism for partitioning the class namespace into more manageable chunks.
- This is called the package

Java package

- It is both a naming and a visibility control mechanism.
- Inside a package Classes can be defined
  - That are not accessible by code outside that package.
  - That are only exposed to other members of the same package.

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

Creating a package

- 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.
- The default package is fine for short and sample programs
- It is inadequate for real applications

Syntax

- **package pkg;**
- Here, pkg is the name of the package.
- For example
  - package MyPackage;
- The .class files for any classes you declare to be part of MyPackage must be stored in a directory called MyPackage.
- **Important**: You cannot rename a package without renaming the directory in which the classes are stored

Package hierarchy

- We can create a hierarchy of packages by separating each package name from the one above it by use of a period.
- The general form: **package pkg1[.pkg2[.pkg3]];**
- A package hierarchy must be reflected in the file system of your Java development system.
- Example:
  - package java.awt.image;
- This package needs to be stored in java\awt\image in a Windows environment.
- The import statement to bring certain classes, or entire packages, into visibility

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

How does the Java run-time system know where you have created packages?

- 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** command to specify the path to your classes

A simple package

```
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);
  }
}
```

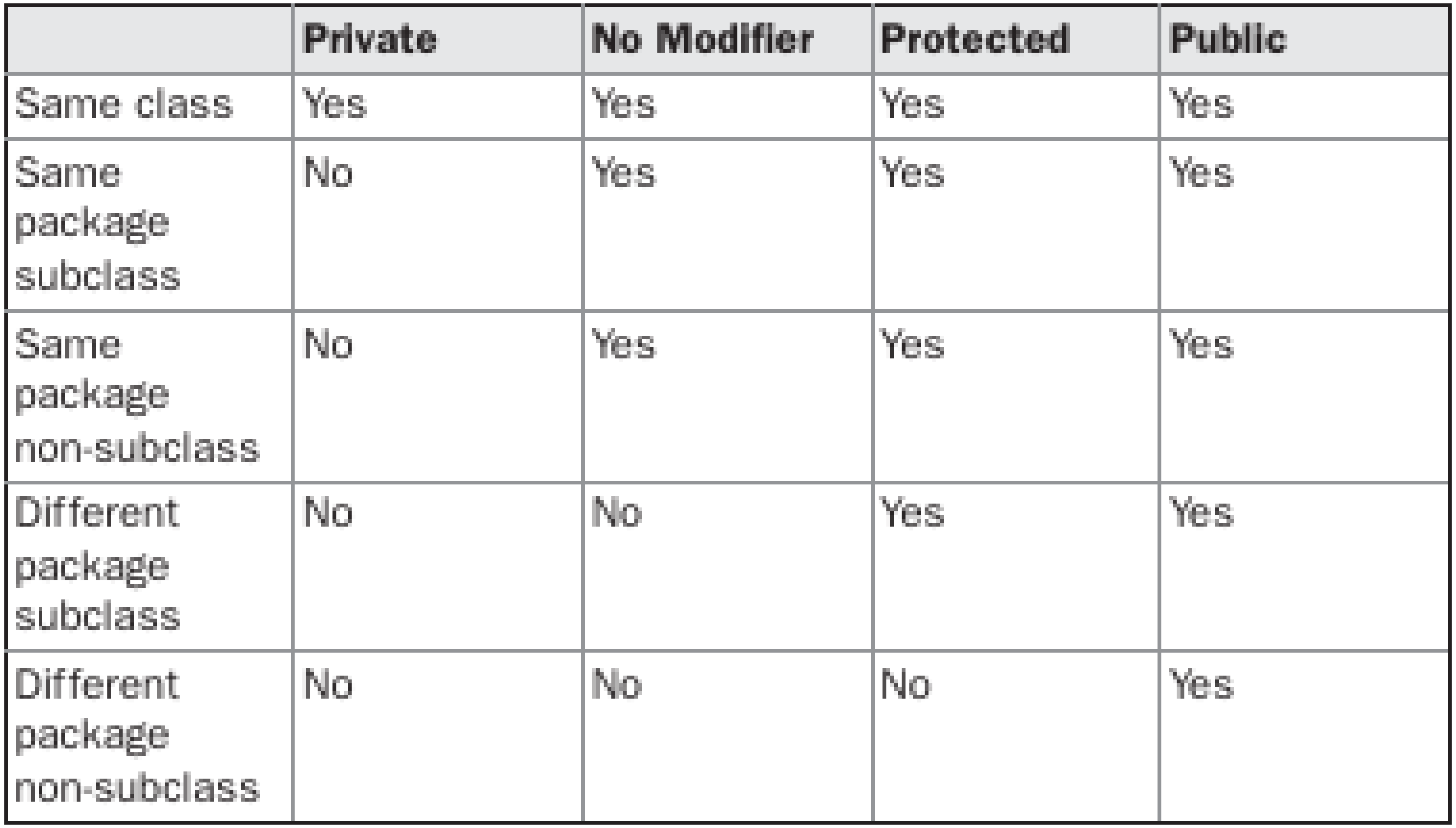
```
class AccountBalance {
  public static void main(String
args[]) {
    Balance current[] = new
Balance[3];

    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();
  }
}
```

Save this file as AccountBalance.java
Put it in a directory called MyPack
Execute AccountBalance class, using **java MyPack.AccountBalance**

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 example

Protection.java

```
package p1;
public class Protection {
        int n = 1;
        private int n_pri = 2;
        protected int n_pro = 3;
        public int n_pub = 4;
        public Protection() {
                System.out.println("Protection base
constructor");
                System.out.println("n = " + n);
                System.out.println("n_pri = " + n_pri);
                System.out.println("n_pro = " + n_pro);
                System.out.println("n_pub = " + n_pub);
        }
}
```

Derived.java

```
package p1;
class Derived extends Protection {
        Derived() {
                System.out.println("derived constructor");
                System.out.println("n = " + n);
        ►       System.out.println("n_pri = " + n_pri);
                System.out.println("n_pro = " + n_pro);
                System.out.println("n_pub = " + n_pub);
        }
}
```

SamePackage.java

```
package p1;
class SamePackage {
        SamePackage() {
                Protection p = new Protection();
                System.out.println("same
                package constructor");
                System.out.println("n = " + p.n);
        ►       System.out.println("n_pri = " + p.n_pri);
                System.out.println("n_pro = " + p.n_pro);
                System.out.println("n_pub = " + p.n_pub);
        }
}
```

Access example(2)

Protection2.java

```
package p2;
class Protection2 extends p1.Protection {
        Protection2() {
                System.out.println("derived other package constructor");
        ►       System.out.println("n = " + n);
        ►       System.out.println("n_pri = " + n_pri);
                System.out.println("n_pro = " + n_pro);
                System.out.println("n_pub = " + n_pub);
        }
}
```

OtherPackage.java

```
package p2;
class OtherPackage {
        OtherPackage() {
                p1.Protection p = new p1.Protection();
                System.out.println("other package constructor");
        ►       System.out.println("n = " + p.n);
        ►       System.out.println("n_pri = " + p.n_pri);
        ►       System.out.println("n_pro = " + p.n_pro);
                System.out.println("n_pub = " + p.n_pub);
        }
}
```

Importing package

- To make a class of the package available as a stand-alone class for general use outside of its package, it needs to declare it as public
- Then, import the classfile of the package in the java program accessing the class using import keyword

Interfaces

- Specifies 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.
- 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 you to fully utilize the “one interface, multiple methods” aspect of polymorphism.

Interfaces(2)

- Methods declared in an interface are always public and abstract (keywords can be safely omitted)
- Static methods cannot be declared in the interfaces – these methods are never abstract and do not express behavior of objects
- Variables can be declared in the interfaces. They can only be declared as static and final. (keywords can be safely omitted)
- Sometimes interfaces declare only constants – be used to effectively import sets of related constants.

Interface vs Abstact class

- An interface is simply a list of unimplemented, and therefore abstract methods.
- An interface cannot implement any methods, whereas an abstract class can.
- A class can implement many interfaces but can have only one superclass.
- An interface is not part of the class hierarchy. Unrelated classes can implement the same interface.

syntax

```
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;
}
```

Implementing an interface

- To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.

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

- The type signature of the implementing method must match exactly the type signature specified in the interface definition.
- 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
- When you implement an interface method, it must be declared as public.

Example

```
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);
  }
}
-------------------------------------------

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

types of interface--2   functional interface and marker interface
functional interface has only one abstract method and rest others   ex. - interface{ void fun(); }
runnable interface
marker interface --- has empty implementation   ex.- cloneable and serializable interface

More about interfaces

- A class automatically implements all interfaces that are implemented by its superclass
- Interfaces belong to Java namespace and as such are placed into packages just like classes.
- Some interfaces are declared with entirely empty bodies. They serve as labels for classes
- One interface can inherit another by use of the keyword **extends**
- The most common marker interfaces are **Cloneable** and **Serializable**

```
class Car implement Cloneable
{ …
        public Object clone()
        {    return super.clone();
        }
}
```

Variables in Interfaces

```
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;
        }
}
```

```
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());
        }
}
```