JAVA PROGRAMMING

Week 4: Packages and interfaces

Lecturer:

- HO Tuan Thanh, M.Sc.

Plan

1. Packages
2. Interfaces

Plan

1. **Packages**
2. Interfaces

Packages

- In programming: it is often helpful to group related pieces of a program together.
- In Java, this can be accomplished by using a package. A package serves two purposes
  - Provides a mechanism by which related pieces of a program can be organized as a unit → A package provides a way to name a collection of classes.
  - A package participates in Java's access control mechanism.
- In Java, no two classes can use the same name from the same namespace → within a given namespace, each class name must be unique.

Defining a Package [1]

- All classes in Java belong to some package.
- When no package statement is specified:
  - The default package is used.
  - The default package has no name, which makes the default package transparent.
- To create a package:
  - Put a package command at the top of a Java source file.
  - The classes declared within that file will then belong to the specified package.
- General form:

```
package pkg;
```

  - pkg is the name of the package.

Defining a Package [2]

- Java uses the file system to manage packages → each package stored in its own directory.
- Package names are case sensitive.
  - This means that the directory in which a package is stored must be precisely the same as the package name.
  - Lowercase is often used for package names.
- 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. General form:

```
package pack1.pack2.pack3...packN;
```

Finding Packages and CLASSPATH

- How does the Java runtime system know where to look for packages that you create?

→By default: the Java runtime system uses the current working directory as its starting point → if your package is in a subdirectory of the current directory, it will be found.

→You can specify a directory path or paths by setting the CLASSPATH environmental variable.

→You can use the classpath option with java and javac to specify the path to your classes.

- From JDK 9: A package can be part of a module, and thus found on the module path.

```
1.  package bookpack;
2.  class Book {
3.      private String title;
4.      private String author;
5.      private int pubDate;
6.      Book(String t, String a, int d) {
7.          title = t;
8.          author = a;
9.          pubDate = d;
10.     }
11.     void show() {
12.         System.out.println(title);
13.         System.out.println(author);
14.         System.out.println(pubDate);
15.         System.out.println();
16.     }
17. }
```

```
1.  package bookpack;
2.  class BookDemo {
3.      public static void main(String args[]) {
4.          Book books[] = new Book[5];
5.          books[0] = new Book("Java: A Beginner's Guide",
6.                                                  "Schildt", 2014);
7.          books[1] = new Book("Java: The Complete Reference",
8.                                                  "Schildt", 2014);
9.          books[2] = new Book("The Art of Java",
10.                                             "Schildt and Holmes", 2003);
11.         books[3] = new Book("Red Storm Rising",
12.                                                 "Clancy", 1986);
13.         books[4] = new Book("On the Road",
14.                                                 "Kerouac", 1955);
15.         for (int i = 0; i < books.length; i++)
16.             books[i].show();
17.     }
18. }
```

PACKAGES AND MEMBER ACCESS

	Private Member	Default Member	Protected Member	Public Member
Visible within same class	Yes	Yes	Yes	Yes
Visible within same package by subclass	No	Yes	Yes	Yes
Visible within same package by non-subclass	No	Yes	Yes	Yes
Visible within different package by subclass	No	No	Yes	Yes
Visible within different package by non-subclass	No	No	No	Yes

```
1.  package bookpack;
2.  //Book recoded for public access.
3.  public class Book {
4.          private String title;
5.          private String author;
6.          private int pubDate;
7.          // Now public.
8.          public Book(String t, String a, int d) {
9.                  title = t; author = a; pubDate = d;
10.         }
11.         // Now public.
12.         public void show() {
13.                 System.out.println(title);
14.                 System.out.println(author);
15.                 System.out.println(pubDate);
16.                 System.out.println();
17.         }
18. }
```

```
1.  //This class is in package bookpackext.
2.  package bookpackext;
3.
4.      //Use the Book Class from bookpack.
5.  class UseBook {
6.      public static void main(String args[]) {
7.          bookpack.Book books[] = new bookpack.Book[5];
8.          books[0] = new bookpack.Book(
9.            "Java: A Beginner's Guide", "Schildt", 2014);
10.         books[1] = new bookpack.Book(
11.           "Java: The Complete Reference", "Schildt", 2014);
12.         books[2] = new bookpack.Book("The Art of Java",
13.           "Schildt and Holmes", 2003);
14.         books[3] = new bookpack.Book("Red Storm Rising",
15.            "Clancy", 1986);
16.         books[4] = new bookpack.Book("On the Road",
17.            "Kerouac", 1955);
18.         for(int i=0; i < books.length; i++) books[i].show();
19.     }
20. }
```

UNDERSTANDING PROTECTED MEMBERS

- The protected modifier creates a member that is accessible within its package and to subclasses in other packages.

→ A protected member is available for all subclasses to use but is still protected from arbitrary access by code outside its package.

```
1.  package bookpack;
2.  //Make the instance variables in Book protected.
3.  public class Book {
4.      // these are now protected
5.      protected String title;
6.      protected String author;
7.      protected int pubDate;
8.
9.      public Book(String t, String a, int d) {
10.         title = t; author = a; pubDate = d;
11.     }
12.     public void show() {
13.         System.out.println(title);
14.         System.out.println(author);
15.         System.out.println(pubDate);
16.         System.out.println();
17.     }
18. }
```

Example [1]


```
1.  package bookpackext;
2.  //Demonstrate Protected.
3.  class ExtBook extends bookpack.Book {
4.      private String publisher;
5.      public ExtBook(String t, String a, int d, String p) {
6.          super(t, a, d); publisher = p;
7.      }
8.      public void show() {
9.          super.show();
10.         System.out.println(publisher);
11.         System.out.println();
12.     }
13.     public String getPublisher() { return publisher; }
14.     public void setPublisher(String p) { publisher = p; }
```

Example [2]


```
1.          /* These are OK because subclass can access
2.                  a protected member. */
3.          public String getTitle() { return title; }
4.          public void setTitle(String t) { title = t; }
5.          public String getAuthor() { return author; }
6.          public void setAuthor(String a) { author = a; }
7.          public int getPubDate() { return pubDate; }
8.          public void setPubDate(int d) { pubDate = d;
9.      }
10. }
```

```
1.  package bookpackext;
2.
3.  class ProtectDemo {
4.      public static void main(String args[]) {
5.          ExtBook books[] = new ExtBook[5];
6.          books[0] = new ExtBook("Java: A Beginner's Guide",
7.                                 "Schildt", 2014, "McGraw-Hill");
8.          books[1] = new ExtBook("Java: The Complete Reference",
9.                                 "Schildt", 2014, "McGraw-Hill");
10.         books[2] = new ExtBook("The Art of Java",
11.                    "Schildt and Holmes", 2003, "McGraw-Hill");
12.         books[3] = new ExtBook("Red Storm Rising",
13.                                "Clancy", 1986, "Putnam");
14.         books[4] = new ExtBook("On the Road",
15.                                "Kerouac", 1955, "Viking");
16.         for (int i = 0; i < books.length; i++)
17.             books[i].show();
```

```
1.              // Find books by author
2.              System.out.println("Showing all books by Schildt.");
3.              for (int i = 0; i < books.length; i++)
4.                      if (books[i].getAuthor() == "Schildt")
5.                              System.out.println(books[i].getTitle());
6.                      // books[0].title = "test title";
7.                      // Error -- not accessible
8.          }
9.      }
```

IMPORTING PACKAGES

- To use a class from another package: you can fully qualify the name of the class with the name of its package → such an approach could easily become tiresome and awkward, especially if the classes you are qualifying are deeply nested in a package hierarchy.
- General form:

import pkg.classname;

  - pkg is the name of the package, which can include its full path,
  - classname is the name of the class being imported.
  - If you want to import the entire contents of a package, use an asterisk (*) for the class name.

```
1.  package bookpackext;
2.  import bookpack.*;
3.  //Use the Book Class from bookpack.
4.  class UseBook {
5.      public static void main(String args[]) {
6.          Book books[] = new Book[5];
7.          books[0] = new Book("Java: A Beginner's Guide",
8.                  "Schildt", 2014);
9.          books[1] = new Book("Java: The Complete Reference",
10.                 "Schildt", 2014);
11.         books[2] = new Book("The Art of Java",
12.                 "Schildt and Holmes", 2003);
13.         books[3] = new Book("Red Storm Rising",
14.                 "Clancy", 1986);
15.         books[4] = new Book("On the Road", "Kerouac", 1955);
16.         for (int i = 0; i < books.length; i++) books[i].show();
17.     }
18. }
```

JAVA'S CLASS LIBRARY IS CONTAINED IN PACKAGES

Subpackage	Description
java.lang	Contains a large number of general-purpose classes
java.io	Contains I/O classes
java.net	Contains classes that support networking
java.util	Contains a large number of utility classes, including the Collections Framework
java.awt	Contains classes that support the Abstract Window Toolkit

Plan

1. Packages
2. **Interfaces**

Interface [1]

- An interface is syntactically similar to an abstract class → you can specify one or more methods that have no body.
- An interface specifies what must be done, but not how to do it.
  - Once an interface is defined, any number of classes can implement it.
  - Also, one class can implement any number of interfaces.
- To implement an interface: a class must provide bodies (implementations) for the methods described by the interface.
  - Each class is free to determine the details of its own implementation.
  - Two classes might implement the same interface in different ways, but each class still supports the same set of methods.

Interface [2]


```
access interface name {
        rettype methodname1(paramlist);
        rettype methodname2(paramlist);
        type var1 = value;
        type var2 = value;
        // ...
        rettype methodnameN(paramlist);
        type varN = value;
}
```

- access is either public or not used
- name is the name of the interface and can be any valid identifier, except for var
- methods are implicitly public

Example


```
1.  public interface Series {
2.      int getNext(); // return next number in series
3.      void reset(); // restart
4.      void setStart(int x); // set starting value
5.  }
```

IMPLEMENTING INTERFACES


```
class classname extends superclass
                        implements interface {
// classbody
}
```

- To implement more than one interface: the interfaces are separated with a comma.
- The extends clause is optional.
- 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.

```
1.  //Implement Series.
2.  class ByTwos implements Series {
3.          int start;
4.          int val;
5.          ByTwos() { start = 0; val = 0; }
6.          public int getNext() {
7.                  val += 2;  return val;
8.          }
9.          public void reset() { val = start; }
10.         public void setStart(int x) {
11.                 start = x;
12.                 val = x;
13.         }
14. }
```

```
1.  class SeriesDemo {
2.      public static void main(String args[]) {
3.          ByTwos ob = new ByTwos();
4.          for (int i = 0; i < 5; i++)
5.            System.out.println("Next value is " + ob.getNext());
6.          System.out.println("\nResetting");
7.          ob.reset();
8.          for (int i = 0; i < 5; i++)
9.            System.out.println("Next value is " + ob.getNext());
10.         System.out.println("\nStarting at 100");
11.         ob.setStart(100);
12.         for (int i = 0; i < 5; i++)
13.           System.out.println("Next value is " + ob.getNext());
14.     }
15. }
```

```
1.  //Implement Series and add getPrevious().
2.  class ByTwos implements Series {
3.          int start;
4.          int val;
5.          int prev;
6.          ByTwos() { start = 0; val = 0; prev = -2; }
7.          public int getNext() {
8.                  prev = val; val += 2; return val;
9.          }
10.         public void reset() {
11.                 val = start; prev = start - 2;
12.         }
13.         public void setStart(int x) {
14.                 start = x; val = x; prev = x - 2;
15.         }
16.         int getPrevious() { return prev; }
17. }
```

Example: Another implementation of the interface


```
1.  //Implement Series.
2.  class ByThrees implements Series {
3.          int start;
4.          int val;
5.          ByThrees() { start = 0; val = 0; }
6.          public int getNext() { val += 3; return val; }
7.          public void reset() { start = 0; val = 0; }
8.          public void setStart(int x) { start = x; val = x; }
9.  }
```

USING INTERFACE REFERENCES

- You can declare a reference variable of an interface type.
- In other words:
  - You can create an interface reference variable.
  - Such a variable can refer to any object that implements its interface.
  - When you call a method on an object through an interface reference, it is the version of the method implemented by the object that is executed.
  - This process is similar to using a superclass reference to access a subclass object.

Example


```
1.  class SeriesDemo2 {
2.      public static void main(String args[]) {
3.          ByTwos twoOb = new ByTwos();
4.          ByThrees threeOb = new ByThrees();
5.          Series ob;
6.          for (int i = 0; i < 5; i++) {
7.              ob = twoOb;
8.              System.out.println("Next ByTwos value is " +
9.                                                  ob.getNext());
10.             ob = threeOb;
11.             System.out.println("Next ByThrees value is " +
12.                                                 ob.getNext());
13.         }
14.     }
15. }
```

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.

```
1.  //One interface can extend another.
2.  interface A {
3.      void meth1();
4.      void meth2();
5.  }
6.  //B now includes meth1() and meth2() -- it adds meth3().
7.  interface B extends A {
8.      void meth3();
9.  }
10. //This class must implement all of A and B
11. class MyClass implements B {
12.     public void meth1() {
13.         System.out.println("Implement meth1().");
14.     }
15.     public void meth2() {
16.         System.out.println("Implement meth2().");
17.     }
18.     // ...
```

```
1.          // ...
2.          public void meth3() {
3.                  System.out.println("Implement meth3().");
4.          }
5.  }
6.
7.  class IFExtend {
8.          public static void main(String arg[]) {
9.                  MyClass ob = new MyClass();
10.
11.                 ob.meth1();
12.                 ob.meth2();
13.                 ob.meth3();
14.         }
15. }
```

DEFAULT INTERFACE METHODS [1]

- Prior to JDK 8:
  - An interface could not define any implementation.
  - The methods specified by an interface were abstract, containing no body.
  - This is the traditional form of an interface.
- From JDK 8:
  - A default method lets you define a default implementation for an interface method.

  → It is 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.

DEFAULT INTERFACE METHODS [2]

- Motivation:
  - Provide a means by which interfaces could be expanded without breaking existing code.
  - Specify methods in an interface that are, essentially, optional, depending on how the interface is used.
- The addition of default methods does not change a key aspect of interface: an interface still cannot have instance variables.
- Default methods constitute a special-purpose feature.
  - Interfaces that you create will still be used primarily to specify what and not how.
  - However, the inclusion of the default method gives you added flexibility.

Default Method Fundamentals

- An interface default method is defined similar to the way a method is defined by a class.
- The primary difference: The declaration is preceded by the keyword default.

Exemple [1]


```
1.  public interface MyIF {
2.      // This is a "normal" interface method declaration.
3.      // It does NOT define a default implementation.
4.      int getUserID();
5.
6.      // This is a default method. Notice that it provides
7.      // a default implementation.
8.      default int getAdminID() {
9.          return 1;
10.     }
11. }
```

Exemple [2]


```
1.  //Implement MyIF.
2.  class MyIFImp implements MyIF {
3.      // Only getUserID() defined by MyIF needs to be
4.      // implemented.
5.      // getAdminID() can be allowed to default.
6.      public int getUserID() {
7.          return 100;
8.      }
9.  }
```

Exemple [3]


```
1.  //Use the default method.
2.  class DefaultMethodDemo {
3.      public static void main(String args[]) {
4.          MyIFImp obj = new MyIFImp();
5.          // Can call getUserID(), because it is explicitly
6.          // implemented by MyIFImp:
7.          System.out.println("User ID is " +
8.                                                  obj.getUserID());
9.          // Can also call getAdminID(), because of default
10.         // implementation:
11.         System.out.println("Administrator ID is " +
12.                                                 obj.getAdminID());
13.     }
14. }
```

Exemple [4]


```
1.  class MyIFImp2 implements MyIF {
2.      // Here, implementations for both getUserID() and
3.      // getAdminID() are provided.
4.      public int getUserID() {
5.          return 100;
6.      }
7.
8.      public int getAdminID() {
9.          return 42;
10.     }
11. }
```

Multiple Inheritance Issues [1]

- Java does not support the multiple inheritance of classes.
- Default methods do offer a bit of what one would normally associate with the concept of multiple inheritance.
- Example:
  - 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 imple

→ Java defines a set of rules that resolve such conflicts.

Multiple Inheritance Issues [2]

- In all cases: a class implementation takes priority over an interface default implementation.
  - →If MyClass provides an override of the reset() default method, MyClass's version is used.
  - →If MyClass implements both Alpha and Beta → both defaults are overridden by MyClass's implementation.
- In cases in which a class inherits two interfaces that both have the same default method: if the class does not override that method, then an error will result.
- In cases in which one interface inherits another, with both defining a common default method, the inheriting interface's version of the method takes precedence.
  - →If Beta extends Alpha, then Beta's version of reset( ) will be used.

Multiple Inheritance Issues [3]

- To refer explicitly to a default implementation by using super.
- General form:

```
InterfaceName.super.methodName()
```

USE STATIC METHODS IN AN INTERFACE

- From JDK 8: It is possible to define one or more static methods.
- A static method defined by an interface can be called independently of any object.
  - →no implementation of the interface is necessary, and no instance of the interface is required in order to call a static method.
  - →A static method is called by specifying the interface name, followed by a period, followed by the method name.
- General form:

```
InterfaceName.staticMethodName
```

- Static interface methods are not inherited by either an implementing class or a sub-interface.

Exemple


```
1.  public interface MyIF {
2.      // This is a "normal" interface method declaration.
3.      // It does NOT define a default implementation.
4.      int getUserID();
5.      // This is a default method. Notice that it provides
6.      // a default implementation.
7.      default int getAdminID() {
8.          return 1;
9.      }
10.     // This is a static interface method.
11.     static int getUniversalID() {
12.         return 0;
13.     }
14. }
```

PRIVATE INTERFACE METHODS

- From JDK 9:
  - An interface can include a private method.
  - A private interface method can be called only by a default method or another private method defined by the same interface.
  - Because a private interface method is specified private, it cannot be used by code outside the interface in which it is defined.
  - This restriction includes sub-interfaces because a private interface method is not inherited by a subinterface.
- Benefit:
  - It lets two or more default methods use a common piece of code, thus avoiding code duplication.

QUESTION ?