Chapter: 5

**Packages and Interfaces**

Packages, Java-Standard Package, and Interfaces

**5.1 Packages (Encapsulation of classes)**

The package provides a way by which classes can be encapsulated. A package serves two purposes:

* Package is a mechanism by which *related pieces of a program can be organized as a unit*. Classes defined within a package must be accessed through their *package name*. A package ***provides a way to name*** a collection of classes.
* A package participates in Java’s *access control mechanism*. Classes defined within a package can be made ***private*** to that package and not accessible by code outside the package.
* Classes and namespaces (recall C/C++ 14.1 ): In general, when you name a class, you are allocating a name from the *namespace*. A *namespace* defines a ***declarative region***. In Java, within a given *namespace*, each class name must be *unique*.
* The examples shown in the preceding chapters have all used the default (global) namespace. While this is fine for short sample programs, it becomes a problem as programs grow and the default namespace becomes crowded.
* Also, you must avoid name collisions with code created by other programmers of same project, and with Java’s library.
* *Package is the solution of the namespace problem:* package gives you a way to ***partition*** the namespace. When a class is defined within a *package*, the name of that *package* is attached to each class, thus avoiding name collisions with other classes that have the same name, but are in other *packages*.
* Since a *package* usually contains *related classes*, Java defines special access rights to code within a package. ***In a package, you can define code that is accessible by other code within the same package but not by code outside the package***. This enables you to create self-contained groups of related classes that keep their operation private.

**5.2 Defining a Package**

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*. Since a package defines a *namespace*, the names of the classes that you put into the file become part of that *package’s namespace*. General form of the package statement:

***package pkg;***

* Here, ***pkg*** is the name of the package. Eg, the statement creates a package called ***mypack***: **package** mypack;
* Java uses the ***file system*** to manage packages, with each package stored in its own directory. For example, the ***.class*** files for any classes you declare to be part of ***mypack*** must be stored in a directory called ***mypack***.
* Package names *are* case sensitive: *directory* in which a *package* is stored must be precisely the same as the *package name*.
* *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*. It does not exclude other classes in other files from being part of that same package. Most real-world packages are *spread across many files*.
* To create a hierarchy of packages: simply separate each package name from the one above it by use of a period. General form:

***package pack1.pack2.pack3...packN;***

* Of course, you must create directories that support the package hierarchy that you create. For example,

**package** alpha.beta.gamma;

must be stored in ***.../alpha/beta/gamma***, where **...** specifies the path to the ***specified directories***.

* Finding Packages and CLASSPATH: Since packages are mirrored by directories then, how does the Java run-time system know where to look for packages that you create? The answer has three parts.

1. 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.
2. Second, you can specify a directory path or paths by setting the ***CLASSPATH*** environmental variable.
3. Third, you can use the ***-classpath*** option with ***java*** and ***javac*** to specify the path to your classes.

* For example, assuming the package specification:

**package** mypack

In order for a program to find ***mypack***, one of three things must be true:

1. The program can be executed from a directory immediately above ***mypack***,
2. or ***CLASSPATH*** must be set to include the path to ***mypack***,
3. or the ***-classpath*** option must specify the path to ***mypack*** when the program is run via java.

* To avoid problems, it is best to keep all ***.java*** and ***.class*** files associated with a *package* in that *package’s directory*. Also, compile each file from the directory above the package directory.

[The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the .class files into the appropriate directories, and then execute the programs from the development directory. This is the approach used by the following examples.]

Note

All classes in Java belong to some package. When no package statement is specified, the *default (global) package* is used. Furthermore, the *default package* has ***no name***, which makes the default package ***transparent***.

* Example of package: It creates a simple book database that is contained within a package called bookpack.

|  |  |
| --- | --- |
| **package** bookpack; */\* This file is part of the bookpack package. \*/*  **class** Book { */\* Thus, Book is part of bookpack. \*/*  **private** **String** title;  **private** **String** author;  **private** **int** pubDate;  **Book**(**String** t, **String** a, **int** d) { title = t;  uthor = a;  pubDate = d; }  **void** show() { **System.out.println**(title);  **System.out.println**(author);  **System.out.println**(pubDate);  **System.out.println**(); }  } | **class** BookDemo { */\* BookDemo is also part of bookpack. \*/*  **public static void main(String args[])** {  **Book** books[] = **new** Book[5];  books[0] = **new** Book("IT", "King", 2014);  books[1] = **new** Book("IQ84", "Murakami", 2014);  books[2] = **new** Book("The Mist", "King", 2003);  books[3] = **new** Book("Carrie", "King", 1986);  books[4] = **new** Book("Pet Cemetery ", "King", 1955);  **for**(**int** i=0; i < books.**length**; i++) books[i].show(); }} |

* Call this file ***BookDemo.java*** and put it in a directory called ***bookpack***.
* Next, compile the file by specifying: **javac bookpack/BookDemo.java** from the directory directly above ***bookpack***.
* Then try executing the class, using the command line: **java bookpack.BookDemo**
* Remember, you will need to be in the directory above ***bookpack*** when you execute this command. (Or, use one of the other two options described in the preceding section to specify the path to ***bookpack***.)
* As explained, ***BookDemo*** and ***Book*** are now part of the package ***bookpack***. This means that ***BookDemo*** ***cannot be executed by itself***. i.e. *command line:* ***java BookDemo*** cannot be used. Instead, ***BookDemo*** must be qualified with its package name.

**5.3 Packages and Access specifier**

The visibility of an element is determined by its access specification—private, public, protected, or default—and the package in which it resides. i.e the visibility of an element is determined by its *visibility within a* ***class*** and its *visibility within a* ***package***. Notice table below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class member MULTILEVEL Access for different Access-specifier | | | | |
|  | 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 |

Default: If a member of a class has no explicit access modifier, then it is visible within its package but not outside its package.

Public: Members explicitly declared ***public*** are visible everywhere, including *different classes* and *different packages*.

Private: A ***private*** member is accessible only to the other members of its *class* and unaffected by its membership in a package.

Protected: A member specified as ***protected*** is accessible within its package and to all subclasses, including subclasses in other packages.

* Above Table applies only to members of classes. A top-level class has only *two possible access levels:* ***default*** and ***public***.
* When a class is declared as ***public***, it is accessible by any other code. And it must reside in a *file* by the *same name*.
* If a class has ***default*** access, it can be accessed only by other code within its *same package*.
* Example: Consider the previous example of ***Book*** and ***BookDemo*** which were in the same package. Now consider ***Book*** is in one package and ***BookDemo*** in another. In this case, access to ***Book*** would be denied.

|  |  |
| --- | --- |
| * To make ***Book*** available to other *packages*, you must make three changes: | * First, ***Book*** needs to be declared ***public***. This makes ***Book*** visible outside of ***bookpack***. * Second, its *constructor* must be made ***public***, and * finally, its ***show()*** method needs to be ***public***. |

* This allows them to be visible outside of ***bookpack***, too.
* To use ***Book*** from another *package*, either you must use the import statement described in the next section, or you must fully qualify its name to include its full package specification. The re-coded version of ***Book*** class is contained in ***bookpack*** package and class called ***UseBook*** is contained in the ***bookpackext*** package:

|  |  |
| --- | --- |
| **package** bookpack;  */\* Book and its members must be public in order to be used by other packages. \*/*  **public** **class** Book {**private** **String** title;  **private** **String** author;  **private** **int** pubDate;  */\* Now public.\*/*  **public** Book(**String** t, **String** a, **int** d) {  title = t;  author = a;  pubDate = d; }  */\* Now public.\*/* **public** **void** show() {  **System.out.println**(title);  **System.out.println**(author);  **System.out.println**(pubDate);  **System.out.println**(); }  } | */\* This class is in package bookpackext.\*/*  **package** bookpackext;  **class** UseBook {  **public static void main(String args[])** {  */\* Use Book class from bookpack. Qualify Book with its package name: bookpack \*/*  **bookpack.Book** books[] = **new** bookpack.Book[5];  books[0] = **new** bookpack.Book("IT", "King", 2014);  books[1] = **new** bookpack.Book("IQ84", "Murakami", 2014);  books[2] = **new** bookpack.Book("The Mist", "King", 2003);  books[3] = **new** bookpack.Book("Carrie", "King", 1986);  books[4] = **new** bookpack.Book("Pet Cemetery ", "King", 1955);  **for**(**int** i=0; i < books.**length**; i++) books[i].show(); }} |

* Notice how every use of ***Book*** is preceded with the ***bookpack*** qualifier. Without this specification, Book would not be found when you tried to compile ***UseBook***.

**5.4 Protected Members in Packages**

Protected modifier creates a member that is ***accessible within its package*** and to *subclasses* in other *packages*. Thus, a protected member is available for all subclasses to use but is still protected from ***arbitrary access by code outside*** its *package*. For example: consider the previous example of ***Book***

* First, change the ***Book*** class so that its instance variables are ***protected***
* Next, create a *subclass* of ***Book***, called ***ExtBook***, and a *class* called ***ProtectDemo*** that uses ***ExtBook***.
* ***ExtBook*** adds a field that stores the *name of the publisher* and several *accessor methods*.
* Both of these classes will be in their *own package* called ***bookpackext***. They are shown here:

|  |  |
| --- | --- |
| **package** bookpack;  */\* Book and its members must be public in order to be used by other packages. \*/*  **public** **class** Book { *// these are now protected*  **protected** **private** **String** title;  **protected** **private** **String** author;  **protected** **private** **int** pubDate;  */\* Now public.\*/*  **public** Book(**String** t, **String** a, **int** d) {  title = t;  author = a;  pubDate = d; }  */\* Now public.\*/* **public** **void** show() {  **System.out.println**(title);  **System.out.println**(author);  **System.out.println**(pubDate);  **System.out.println**(); }  } | package bookpackext;  **class** ExtBook **extends** ***bookpack.Book*** { */\* Use Book class from bookpack.\*/*  **private** **String** publisher;  **public** ExtBook(**String** t, **String** a, **int** d, **String** p){ super(t, a, d);  publisher = p; }  **public** **void** show() {super.show();  System.out.println(publisher,"\n"); }  **public** **String** getPublisher() { **return** publisher; }  **public** **void** setPublisher(**String** p) { publisher = p; }  */\* Following are OK because subclass can access a protected member. \*/*  **public** **String** getTitle() { **return** title; }  **public** **void** setTitle(**String** t) { title = t; }  **public** **String** getAuthor() { **return** author; }  **public** **void** setAuthor(**String** a) { author = a; }  **public** **int** getPubDate() { **return** pubDate; }  **public** **void** setPubDate(**int** d) { pubDate = d; } } |
| **class** ProtectDemo { **public static void main(String args[])** {  **ExtBook** books[] = **new** ExtBook[5];  books[0] = **new** ExtBook("Java: A Beginner's Guide", "Schildt", 2014, "McGraw-Hill Professional");  */\* Access to Book’s members \*/* books[1] = **new** ExtBook("Java: The Complete Reference", "Schildt", 2014, "McGraw-Hill Professional");  */\* is allowed for subclasses. \*/* books[2] = **new** ExtBook("The Art of Java", "Schildt and Holmes", 2003, "McGraw-Hill Professional");  books[3] = **new** ExtBook("Red Storm Rising", "Clancy", 1986, "Putnam");  books[4] = **new** ExtBook("On the Road", "Kerouac", 1955, "Viking");    **for**(**int** i=0; i < books.**length**; i++) books[i].show();  *// Find books by author*  **System.out.println**("Showing all books by Schildt.");  **for**(**int** i=0; i < books.**length**; i++) **if**(books[i].getAuthor() == "Schildt") **System.out.println**(books[i].getTitle());  ***/\* books[0].title = "test title"; \*/*** *// Error – not accessible*  }} | |

* Because ***ExtBook*** extends ***Book***, it has access to the *protected members* of ***Book***, even though ***ExtBook*** is in a different package. Thus, it can access ***title***, ***author***, and ***pubDate*** directly, as in the *accessor methods* it creates for those variables.
* However, in ***ProtectDemo***, access to these ***title***, ***author***, and ***pubDate*** variables is denied because ***ProtectDemo*** is not a subclass of ***Book***. For example, because of the following line, the program will not compile.

**books[0].title = "test title";** *// Error – not accessible*

Note

Access specifier called ***protected*** in C++ is Similar, but not the same: In **C++**, ***protected*** creates a member that can be accessed by *subclasses* (only) but is otherwise ***private***.

* In **Java**, ***protected*** creates a member that can be accessed by ***any code within its*** *package**(including subclasses inside that package)* but only by *subclasses*outside ***of its*** *package*.

**5.5 Importing Packages**

When you use a ***class*** from another ***package***, you can fully qualify the *name of the class* with the *name of its package*, as the preceding examples (see 5.3 Example) have done.

* However, the shortest and easy way is: using the ***import*** statement. Using ***import*** you can bring one or more members of a package into view. This allows you to use those members directly, without explicit package qualification.
* The general form of the import statement: **import pkg.classname;**
* Here, ***pkg*** is the name of the package, which can include its ***full path***, and ***classname*** is the name of the class being imported.
* To import the entire contents of a package, use an asterisk (**\***) for the ***class name*** (i.e. after period): **import pkg.\*;**

|  |  |
| --- | --- |
| * Here are examples of both forms: | **import** mypack.MyClass; Here the ***MyClass*** class is imported from ***mypack***.  **import** mypack.\*; Here all of the classes in ***mypack*** are imported. |

* In a Java source file, ***import*** ***statements*** occur immediately following the ***package*** ***statement*** (if it exists) and before any ***class definitions***.

|  |  |
| --- | --- |
| * Eg: UseBook class re-coded using: import bookpack.\*; | **package** bookpackext;  **import** bookpack.\*; *// Import bookpack.*  **class** UseBook { **public static void main(String args[])** *{/\* same code as Example of 5.3\*/*} |

* Notice, no longer need to qualify ***Book*** with its package name. Now, you can refer to ***Book*** directly, without qualification.

**5.6 API: Java's Standard Packages** (Java’s Class Library Is Contained in Packages)

API: Java defines a large number of standard classes that are available to all programs. This class library is often referred to as the Java API ***(Application Programming Interface)***. The Java API is stored in packages.

* At the top of the package hierarchy is ***java***. Descending from ***java*** are several *subpackages*. Here are a few examples:

|  |  |  |
| --- | --- | --- |
| * Actually we've been using ***java.lang*** from beginning. It contains, among several others, the *System class*, which we've been using when performing output using ***println()***. * The ***java.lang*** package is unique because it is imported automatically into every Java program. That's why we didn't import ***java.lang*** in the preceding sample programs. * However, you must explicitly import the other 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.applet*** | Contains classes for creating applets |
| ***java.awt*** | Contains classes that support the Abstract Window Toolkit |

**5.7 More abstraction with Interfaces**

*Interface* is an extended version of *class abstraction*. Thus, an abstract method specifies the *interface* (i.e. *signature* ) to the method but not the implementation. Subclass must provide its own implementation of each *abstract* *method* defined by its *superclass*.

* Interface: In Java, you can fully separate a class’ interface from its implementation by using the keyword ***interface***. An *interface* is syntactically similar to an *abstract class*, in that you can specify one or more ***methods that have no body***. Those methods must be implemented by a class in order for their actions to be defined. By Interface we apply “one interface, multiple methods”.
* 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.
* Prior to ***JDK 8***, an interface could define only what, but not how. Today, it is possible to add a default implementation to an interface method. We will begin by discussing the interface in its traditional form. The default method is described later.
* Here is a simplified general form of a traditional interface:

**access** **interface** name { **ret-type** method-name1(param-list);

**ret-type** method-name2(param-list);

**type** var1 = value;

**type** var2 = value;

// ...

**ret-type** method-nameN(param-list);

**type** varN = value; }

* Here, ***access*** is either public or not used.
* When ***no access modifier*** is included, then ***default access*** results, and the interface is available only to other members of its package.
* When it is declared as ***public***, the interface can be used by any other code. (When an interface is declared public, it must be in a file of the same name.)
* ***name*** is the name of the interface and can be any valid identifier.
* In the traditional form of an ***interface***, ***methods*** are declared using only their *return type* and *signature* (abstract methods).
* In an interface, methods are implicitly ***public***.
* Variables declared in an interface are not instance variables. Instead, they are implicitly ***public***, ***final***, and ***static*** and must be initialized. Thus, they are essentially *constants*.
* Example: Here is an example of an *interface definition*. It specifies the interface to a class that generates a series of numbers.

public interface Series { int getNext(); *// return next number in series*

void reset(); *// restart*

void setStart(int x); */\* set starting value \*/* }

* This interface is declared ***public*** so that it can be implemented by code in *any package*.

**5.8 Implementing Interfaces**

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:

***class classname extends superclass*** implementsinterface ***{*** */\* class-body\*/* ***}***

* To implement ***more than one interface***, the interfaces are separated with a comma. And of course, 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***.
* Example: Here is an example that implements the ***interface Series*** shown earlier. It creates a class called ***ByTwos***, which generates a series of numbers, each two greater than the previous one.

|  |  |
| --- | --- |
| **public** **interface** Series {  **int** getNext(); *// return next number in series*  **void** reset(); *// restart*  **void** setStart(int x); */\* set starting value \*/* } | **class** ByTwos **implements** Series { **int** start, val;  ByTwos() { start = 0; val = 0; }  */\* Implement the \*/* **public** **int** getNext() { val += 2; **return** val; }  */\* Series interface. \*/* **public** **void** reset() { val = start;}  **public** **void** setStart(**int** x) { start = x; val = x;} } |

* Notice, ***getNext()***, ***reset()***, and ***setStart()*** are declared as ***public***. This is necessary. Whenever you implement a method defined by an interface, it must be implemented as ***public*** because all members of an interface are implicitly ***public***.
* Here is a class that demonstrates ***ByTwos***:

**class** SeriesDemo { **public static void main(String args[])** { **ByTwos** ob = **new** ByTwos();

**for**(**int** i=0; i < 5; i++) **System.out.println**("Next value is " + ob.getNext());

ob.reset(); *//Resetting*

ob.setStart(100); */\* Starting at 100* *\*/* }}

* It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of ***ByTwos*** adds the method ***getPrevious()***, which returns the previous value:

**class** ByTwos **implements** Series{ **int** start, val, prev;

ByTwos() { start = 0; val = 0; prev = -2; }

**public** **int** getNext(){ prev = val; val += 2; **return** val; }

**public** **void** reset() { val = start; prev = start - 2; }

**public** **void** setStart(**int** x){ start=x; val=x; prev = x-2; }

int getPrevious() { return prev; } *//not defined by Series*

}

* Notice that the addition of ***getPrevious()*** required a change to the implementations of the methods defined by ***Series***. However, since the interface to those methods stays the same, the change is seamless and does not break *preexisting code*.

|  |  |
| --- | --- |
| * As explained, ***any number of classes can implement an interface***. For example, here is a class called ***ByThrees*** that generates a series that consists of multiples of three: | **class** ByThrees **implements** Series { **int** start, val;  ByThrees() { start = 0; val = 0; }  **public** **int** getNext() { val += 3; **return** val; }  **public** **void** reset() { val = start; }  **public** **void** setStart(**int** x) { start = x; val = x; } } |

Note:

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. *No objects of such a class can be created*, but it can be used as an abstract superclass.

**5.9 Using Interface References**

You can declare a *reference variable* of an interface type. i.e, 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: Following uses the same interface reference variable to call methods on objects of both ***ByTwos*** and ***ByThrees***.

|  |  |
| --- | --- |
| **class** ByTwos **implements** Series { **int** start, val;  ByTwos() { start = 0; val = 0; }  **public** **int** getNext() { val += 2; **return** val; }  **public** **void** reset() { val = start; }  **public** **void** setStart(**int** x) { start = x; val = x; } }  **class** ByThrees **implements** Series { **int** start, val;  ByThrees() { start = 0; val = 0; }  **public** **int** getNext() { val += 3; **return** val; }  **public** **void** reset() { val = start; }  **public** **void** setStart(**int** x) { start = x; val = x; } } | **class** SeriesDemo2 {  **public static void main(String args[])** {  **ByTwos** twoOb = **new** ByTwos();  **ByThrees** threeOb = **new** ByThrees();  **Series** ob; *//interface reference variable*  *// Access an object via an interface reference*  for(int i=0; i < 5; i++) { ob = twoOb;  **System.out.println**("Next ByTwos value is " + ob.getNext());  ob = threeOb;  **System.out.println**("Next ByThrees value is " + ob.getNext()); }  } } |

* In ***main()***, ***ob*** is declared to be a reference to a ***Series*** ***interface***. This means that it can be used to store references to any object that *implements* ***Series***. In this case, it is used to refer to ***twoOb*** and ***threeOb***, which are objects of type ***ByTwos*** and ***ByThrees***, respectively, which both *implement* ***Series***.
* An interface reference variable has knowledge only of the ***methods declared by its interface declaration***. Thus, ***ob*** *could not be used to access any other variables or methods* that might be supported by the object *(not the reference variable)*.

**5.10 Variables in Interfaces**

In an *interface variables* are implicitly ***public***, ***static***, and ***final***. Since a large program is typically held in a number of separate *source files*, there needs to be a convenient way to make these constants available to each file. *Interface variable* is a solution.

* To define a *set of shared constants*, create an interface that contains only these constants, ***without any methods***. Each file that needs access to the constants simply “*implements*” the ***interface***. For example:

|  |  |
| --- | --- |
| *// An interface that contains constants.*  **interface** IConst {  **int** MIN = 0;  **int** MAX = 10;  **String** ERRORMSG = "Boundary Error";  } *//These are constants.* | **class** IConstD **implements** IConst {  **public static void main(String args[])** {  **int** nums[] = **new** **int**[MAX];  **for**(int i=MIN; i < 11; i++) {  **if**(i >= MAX) **System.out.println**(ERRORMSG);  **else** { nums[i] = i; **System.out.print**(nums[i] + " "); }  } }} |

NOTE

The technique of using an interface to define shared constants is controversial. It is described here for completeness.

**5.11 Interfaces Can Be Extended**

***One interface can inherit another interface*** 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***. (i.e. including methods that are inherited from other interfaces.). Following is an example:

|  |  |
| --- | --- |
| **interface** A { **void** meth1();  **void** meth2(); }  */\* B inherits A. 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()."); }  } |

* If we remove the implementation for ***meth1()*** in ***MyClass***. This will cause a compile-time error.

**5.12 Default Interface Methods**

Prior to JDK 8, an interface could not define any implementation. i.e. All the methods specified by an interface were abstract, containing no body. In JDK 8 changed default method is introduced.

* Default method: A *default method* lets you define a default implementation for an interface method (having a body rather than empty). The *default method* was also referred to as an *extension method*. Following are reasons for default method:
* Default methods help to extend an interface without breaking existing code: Prior default method problem occurs if a new method were added to a popular, widely used *interface*, then the addition of that method would break pre-existing code because no implementation would be found for that method. The default method solves this problem by supplying an implementation that will be used if no other implementation is explicitly provided.
* *Default methods* allow us to specify methods in an ***interface*** that are, essentially, optional, depending on how the interface is used.

Note

1. An *interface* still cannot have *instance variables*. Thus, the defining difference between an *interface* and a *class* is that a *class* can maintain state information, but an *interface* cannot.
2. It is still not possible to create an ***instance of an interface by itself***. It must be implemented by a class.
3. Interfaces that you create will still be used primarily to specify what and not how.

**5.13 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***. To define a *default method*, precede its declaration with ***default***. For example:

**public** **interface** MyIF { **int** getUserID(); *// This is a "normal" interface method declaration.*

**default** **int** getAdminID() { **return** 1; } *// a default implementation.*

}

* ***getUserID()***, is a *standard interface method declaration*. It defines *no implementation*.
* ***getAdminID()*** include a ***default*** *implementation*. Its declaration is preceded by the ***default*** modifier. It returns ***1***.
* If an implementing class does not provide its own implementation, the default is used. For example,

**class** MyIFImp **implements** MyIF { *// getAdminID() can be allowed to default.*

**public** **int** getUserID() { **return** 100; } *// Only getUserID() defined by MyIF needs to be implemented.*

}

* An implementing class can define its own implementation of a default method. Eg: ***MyIFImp2*** overrides ***getAdminID()***,

**class** MyIFImp2 **implements** MyIF{ *// Here, implementations for both getUserID( ) and getAdminID( ) are provided.*

**public int** getUserID(){ **return** 100;} *// normal interface method*

**public int** getAdminID() { **return** 42; } *// overriding default interface method*

}

* Now, when ***getAdminID()*** is called, a value other than its ***default*** is returned.
* Example: **main()** class: The following code creates an instance of ***MyIFImp*** and uses it to call both ***getUserID()*** and ***getAdminID()***.

**class** DefaultMethodDemo { **public static void main(String args[])** {

**MyIFImp** obj = **new** MyIFImp();

*// Can call getUserID(), because it is explicitly implemented by MyIFImp:*

**System.out.println**("User ID is " + obj.getUserID());

*// Can also call getAdminID(), because of default implementation:*

**System.out.println**("Administrator ID is " + obj.getAdminID()); }}

* The default implementation of ***getAdminID()*** was automatically used. It was not necessary for ***MyIFImp*** to define it.

[Thus, for getAdminID(), implementation by a class is optional. (Of course, its implementation by a class will be required if the class needs to return a different ID.)]

**5.14 Multiple inheritance and Interface**

Although, Java doesn't support multiple inheritance, but we can apply some multiple inheritance through *Interface* and *default interface methods*. For example, you might have a class that *implements* two *interfaces*. If each of these interfaces provides *default methods*, then some behavior is inherited from both. Thus, to a limited extent, *default methods* do support ***multiple inheritance*** of behavior. In such a situation, it is possible that a ***name conflict*** will occur. Eg: say two interfaces Alpha and Beta are implemented by a class 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 the method? |

* To handle these and other similar types of situations, Java defines a set of rules that resolve 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***’s version is used. This is the case even if ***MyClass*** implements both ***Alpha*** and ***Beta***. In this case, both defaults are overridden by ***MyClass***’s *implementation*.
* Second, 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. Continuing with the example, if ***MyClass*** inherits both ***Alpha*** and ***Beta***, but does not override ***reset()***, then an error will occur.
* In cases in which one *interface* inherits another, with a *common* *default* method, the ***inheriting interface’s version*** of the method takes precedence. i.e If ***Beta*** extends ***Alpha***, then ***Beta’s*** version of ***reset()*** will be used.
* It is possible to refer explicitly to a ***default*** implementation by using a new form of ***super***. Its general form is shown here:

**InterfaceName.super.methodName( )**

* + For example, if ***Beta*** wants to refer to ***Alpha***’s default for ***reset()***, it can use: Alpha.**super**.reset();

**5.15 static Methods in an Interface**

We can 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***.

|  |  |
| --- | --- |
| * No implementation of the interface is necessary | * 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**

Notice that this is similar to the way that a static method in a class is called.

* The following shows an example of a *static method* in an *interface* by adding one to ***MyIF***, shown earlier. The static method is ***getUniversalID()***. It returns zero.

**public** **interface** MyIF {

**int** getUserID(); *// "normal" interface method*

**default** **int** getAdminID() { **return** 1; } *// default method*

**static** **int** getUniversalID() { **return** 0; } *// static interface method.*

}

* The ***getUniversalID()*** method can be called, as: **int** uID = MyIF.***getUniversalID()***;
  + Here no *implementation* or *instance* of ***MyIF*** is required to call ***getUniversalID()*** because it is ***static***.

Note:

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