Packages and Access Specifiers

The process of software design is largely a process of organizing. The previous three chapters explored the object-oriented ways you can organize a Java program. This chapter discusses an additional way to organize Java programs that has nothing to do with object-orientation: packages. In Java, a *package* is a library of types (classes and interfaces). This chapter describes four ways to think about packages and shows how to make use of packages in your Java programs.

Once you know about packages, you can understand all the access levels (such as private, protected, etc.) available to types and their fields and methods. This chapter compares all the access levels and gives advice on how to use them.

Avoiding Name Conflicts

The first way to think about packages is as a tool to help you reduce the likelihood of name conflicts in your programs. When you design a Java program, you model the problem domain by identifying and defining types and assigning each a name. Types refer to each other by name, so each type name you assign must be unique. If you design a large program, or incorporate types named and defined by others, you may encounter name conflicts. To address the problem of name conflicts, you use packages.

Packages effectively lengthen type names, making the names more distinctive. In a Java program, every type belongs to some package. A package is a set of types grouped together under a common package name. Each type has a *simple name*, and each package has a *package name*. The name of the package containing a type, plus a dot, plus the type's simple name is the type's *fully qualified name*. For example, if you have a class named CoffeeCup in a package named dishes, "dishes.CoffeeCup" is its fully qualified name. ("dishes" is its package name; "CoffeeCup" its simple name.) The fully qualified name of a type, which is longer and more distinctive than its simple name, enables like-named types from different packages to be used in the same program.

If you discard the package name from a type's fully qualified name, you get the type's simple name. Therefore, the simple name of dishes.CoffeeCup is, simply, CoffeeCup. To useCoffeeCup, types in the same package can just use its simple name. Types in other packages, however, must also identify dishes, the package containing CoffeeCup, as well as its simple name. This ensures that a different CoffeeCup class defined in a different package will not conflict with dishes.CoffeeCup.

To help make type names even more distinctive, you can organize your packages hierarchically. Packages can contain not only types, but other packages as well. The entities contained in a package--its classes, interfaces, and sub-packages--are called the package's *members*.

The fully qualified name of a class nestled deep down inside several packages is the name of each package and the class's simple name, all separated by dots. For instance, if you placedCoffeeCup inside package dishes and placed dishes inside package vcafe (for virtual cafe), the fully qualified name of CoffeeCup would be "vcafe.dishes.CoffeeCup." The greater the number of nested packages in which you place a class, the more dot-separated names the class will have in its fully qualified name, and the more distinctive that fully qualified name will be.

Packages help you guard against the potential of name conflicts in your Java programs. Instead of worrying that the simple name of every type you need to use in a program is unique, you need only worry that every fully qualified name is unique.

One other way to deal with name conflicts involves class loaders and the multiple name spaces offered by the JVM. This will be discussed in Chapter 20.

Hierarchical Organization

A second way to think about packages is as a tool to help you organize the types you create for your program. With packages, you can organize a program into logically related groups of types, and organize the groups hierarchically.

The package is an organizational tool independent of any object-oriented organization of a program. For example, all the types in a particular family of types could belong to the same package, or be spread out across several packages. A class in one package can subclass a class in another package. The only requirement is that the subclass must specify the name of the package containing its superclass as well as the superclass's simple name. When you organize your types into packages, what you are actually organizing is type *names*.

Although you can grant special privileges between types that belong to the same package, a topic that will be discussed later in this chapter, you can't grant special privileges between a types in a package and types in a sub-package. To the types defined in a parent package, a sub-package is just like any other package. From the perspective of a Java compiler or the Java Virtual Machine, nested packages are not really seen as a hierarchy. They are just seen as a set of independent packages, each with a unique name. Packages are seen as a conceptual hierarchy only from the perspective of developers, who can use the hierarchy to express conceptual relationships between different groups of types.

Often, Java compilers and Java Virtual Machines expect the source files or class files contained in a hierarchy of packages to be located in a corresponding directory hierarchy, in which each directory takes the name of a package. Here, the compiler or Java Virtual Machine is using the package hierarchy as a way to locate files on a disk. The actual manner in which a particular compiler or Java Virtual Machine finds class files is a detail specific to each individual development environment or Java Platform implementation. The process of using directory hierarchies that map to package hierarchies to locate class files will be discussed further later in this chapter.

Libraries

A third way to think about packages is simply as libraries. Any Java program you write will make use of libraries developed by others and made available to your program as packages. Any program will at least use the run-time libraries of the Java API, some of which are java.lang, java.io, java.util, java.net, java.awt, and java.applet. If, rather than developing a complete program, you wish to develop class library that other developers can use in their programs, your end-product will be a package.

Hiding the Implementation

The fourth way to think about packages is as a tool that can help you separate interface from implementation. You can grant special access privileges between types within the same package, and you can declare entire types to be accessible only to other types within the same package. The full details of how to do this will be given later in this chapter as part of a discussion of Java's access levels.

Recommended Package Naming Convention

Because the packages used by a program can come from many sources, it is important that you name your packages in a way that won't conflict with the names of packages developed by others. Of course, you don't know what packages might be developed by other programmers, nor how they will name those packages. This points out that the mechanism of packages doesn't actually solve the name conflict problem, it only reduces the likelihood of an actual conflict. Just because you go to the trouble of enclosing your CoffeeCup class in two nested packages--vcafe and dishes--doesn't mean someone else won't inadvertently do the same.

To combat the potential of name conflicts between types developed by different software vendors, Java comes with a recommended naming convention for packages. If everyone would follow the recommended convention when naming their packages, harmony would cover the Earth. Java does not, however, enforce any naming convention, so name conflicts are still possible. It is up to you to do your part in preventing naming conflicts within Java programs.

The official recommendation on package naming is to use the reversed internet domain name of your company or organization as the first part of your package names. Because internet domain names are globally unique, this improves the chances your package names will be globally unique. If your company's domain name were artima.com, for instance, you would start any package name with "com.artima." The fully qualified name of CoffeeCup would become com.artima.vcafe.dishes.CoffeeCup.

All the packages you create must be given a name that will be unique across the scope in which they will be visible. If they will be visible only locally, you needn't use the recommended naming convention. If you are certain your package names are not going to be visible on a global scale, but will remain inside, say, your division, you can devise and follow a division-wide package naming scheme. For any other package, however, following the recommended naming scheme makes you a good Java citizen.

The Package Declaration

As you write a Java program, you must place every class you define into a package, and give each package a unique name. You place a class into a package by including a package declaration at the top of the source file. A *package declaration* is just the keyword package followed by the package name and a semicolon. The package declaration must appear in the source file before any class or interface declaration, and each source file can contain only one package declaration. For example, you would place CoffeeCup into the packagecom.artima.vcafe.dishes as follows:

// In Source Packet in file

// packages/ex1/com/artima/vcafe/dishes/CoffeeCup.java

package com.artima.vcafe.dishes;

public class CoffeeCup {

public static final int MAX\_SHORT\_ML = 237;

public static final int MAX\_TALL\_ML = 355;

public static final int MAX\_GRANDE\_ML = 473;

public void add(int amountOfCoffee) {

System.out.println("Adding " + amountOfCoffee

+ " ml of coffee.");

}

//...

}

The package name in the example above, com.artima.vcafe.dishes, indicates that dishes is a sub-package of vcafe, which is a sub- package of artima, which is a sub-package of com. You needn't have any source file in your program that declares the com package, the com.artima package, or the com.artima.vcafe package. The package statement in the example above is enough to establish the existence of all four packages: com, artima, vcafe, and dishes.

On the other hand, if you do have source files that declare classes as members of, say, the com.artima.vcafe package, those classes have no special relation to the classes of com.artima.vcafe.dishes, as far as the Java language is concerned. To the Java language, com.artima.vcafe and com.artima.vcafe.dishes are just two different packages with two different names. To you, the programmer, however, the hierarchical relationship between the two packages would have meaning: it would express the conceptual relationship between two different groups of types. [bv: is this redundant with something that came before?]

Location of Source and Class Files

Although the location of source and class files for package members at both compile-time and run- time depends on your particular development and runtime environments, many environments require that you create a hierarchy of directories that correspond to the hierarchy of packages. If you were to work on such a system, you would likely have to put the source and class file for the CoffeeCup class defined above in a directory named ".../com/artima/vcafe/dishes" or "...\com\artima\vcafe\dishes", depending on your preferred direction of slash.

To give one concrete example, imagine you are using Sun's JDK 1.1.1 to run a Java program on Microsoft Windows95. You would set an environment variable, CLASSPATH, to indicate to the Java Virtual Machine where it should look for class files. If your CLASSPATH is set to ".;C:\MYLIB;C:\JDK1.1.1\LIB\CLASSES.ZIP", then the compiler and the Java Virtual Machine would look in three places for the classes needed by your program:

1. the current directory, "."
2. a directory named "C:\MYLIB"
3. a zip file named "C:\JDK1.1.1\LIB\CLASSES.ZIP"

If you use com.artima.vcafe.dishes.CoffeeCup in the program, the Java Virtual Machine would first look for a directory, relative to the current directory, named.\com\artima\vcafe\dishes. (It would look here first because "." is the first directory in the CLASSPATH.) If it finds a CoffeeCup.class in that directory, it would load it. If this directory didn't exist, or there was no CoffeeCup.class in that directory, the Java Virtual Machine would look for a directory named C:\MYLIB\com\artima\vcafe\dishes. If it finds a CoffeeCup.classhere, it would load it. Otherwise it would look inside the zip file for a com\artima\vcafe\dishes\CoffeeCup.class. It is unlikely that CoffeeCup.class it is in the zip file, because this is where all the runtime libraries of the Java Platform are kept in JDK 1.1.1.

As it searches through the directories and zip files listed in the class path, the Java Virtual Machine loads the first class file that it encounters with a name that matches the class name,CoffeeCup.class, and a relative directory that matches the package name, com\artima\vcafe\dishes. Once it has loaded the class file, the virtual machine checks the binary data to verify that the class is indeed   com.artima.vcafe.dishes.CoffeeCup.

This Windows95 and JDK 1.1.1 example was just one possible way a Java Platform implementation could locate class files. To find out how your particular Java Platform or Java development environment locates class files, you must consult its documentation.

The Unnamed Package

In every Java program there can be one *unnamed package*, which is simply a package with no name. In a sense, the unnamed package really does have a name, just a very short one, which distinguishes it from the other packages in your program. To place a class into the unnamed package, just define the class in a source file with no package statement. All types declared in this book prior to this chapter were in the unnamed package.

You should not use the unnamed package for a general-purpose library, because it is probably the most common package name used by Java programmers. (In addition, types declared in the unnamed package are accessible only to each other. In other words, a type in a named package can't access a type in the unnamed package.) In general, you will want to partition large Java programs into named packages to better organize your program and to take advantage of the implementation-hiding capabilities of packages. The unnamed package is convenient and appropriate for the core types that make up an applet or application.

The Import Statement

In a Java source file, you have two ways to refer to a class or interface defined in another package. You can either use the fully qualified name of the class everywhere you refer to it, or you can import that class's fully qualified name into your source file and then just use the simple name everywhere. *Importing* a type into a source file means making the compiler recognize the type in that source file by its simple name.

You can't import packages, just types. Import doesn't include any code, like #include of C or C++. It only means that you can use the simple name of a type instead of the fully qualified names.

As an example, imagine you are writing code in the unnamed package that takes advantage of the CoffeeCup class defined in package com.artima.vcafe.dishes. One approach is to just use the fully qualified name of CoffeeCup everywhere, as in:

// In Source Packet in file packages/ex1/Example1a.java

// Deep in the heart of the unnamed package...

class Example1a {

public static void main(String[] args) {

com.artima.vcafe.dishes.CoffeeCup cup =

new com.artima.vcafe.dishes.CoffeeCup();

cup.add(com.artima.vcafe.dishes.CoffeeCup.MAX\_SHORT\_ML);

}

}

This approach is reasonable if the source file has only a few references to a class, but otherwise can make your code tiresome for you to type and others to read. The alternative is to import the class into the source file and then refer to the class by its simple name. Here's an example:

// In Source Packet in file packages/ex1/Example1b.java

// At the top of a file in the unnamed package, import the class.

import com.artima.vcafe.dishes.CoffeeCup;

// Everywhere else in the file, just use the simple name.

class Example1b {

public static void main(String[] args) {

CoffeeCup cup = new CoffeeCup();

cup.add(CoffeeCup.MAX\_TALL\_ML);

}

}

If you find yourself using several types from a single package, you can import all their names from a package into your source file with one import statement by using an asterisk in place of the class or interface name. (Actually, the asterisk only imports classes and interfaces declared as public, a feature that will be described in detail later in this chapter.):

// In Source Packet in file packages/ex1/Example1c.java

// Import all public types from the com.artima.vcafe.dishes package.

import com.artima.vcafe.dishes.\*;

// Everywhere else in the file, just use the simple names.

class Example1c {

public static void main(String[] args) {

CoffeeCup cup = new CoffeeCup();

cup.add(CoffeeCup.MAX\_GRANDE\_ML);

}

}

Import statements such as the ones shown in the examples above reduce the amount of typing required to use types from other packages, but they also make it possible for names to conflict again. For instance, if you imported two different CoffeeCup classes from two different packages, just referring to "CoffeeCup" would be ambiguous. The compiler wouldn't know whichCoffeeCup you were talking about. In this case you would need to explicitly indicate which CoffeeCup you meant by prefacing the simple name with the package name. In other words, even though you imported both CoffeeCup classes, you'll still have to use the fully qualified names to resolve the ambiguity.

As an example, imagine you imported all the public types from two packages, com.artima.vcafe.dishes and com.artima.pencilholders, both of which contained a CoffeeCup class. To use either version of CoffeeCup you would have to use its fully qualified name, as shown below:

// In Source Packet in file

// packages/ex1/com/artima/pencilholders/CoffeeCup.java

package com.artima.pencilholders;

public class CoffeeCup {

public void add(int amountOfPencils) {

System.out.println("Adding " + amountOfPencils

+ " pencils.");

}

//...

}

// In Source Packet in file packages/ex1/Example1d.java

// All types defined in both packages are

// imported, yielding two different classes named "CoffeeCup."

import com.artima.pencilholders.\*;

import com.artima.vcafe.dishes.\*;

class Example1d {

public static void main(String[] args) {

// Somewhere later in the code, you wish to instantiate a

// new CoffeeCup from the com.artima.vcafe.dishes package:

com.artima.vcafe.dishes.CoffeeCup myCoffee =

new com.artima.vcafe.dishes.CoffeeCup();

// While you sip your coffee with the cup from the virtual

// cafe, you also want a place to store your spare pencils.

// So, you create a new CoffeeCup from the

// com.artima.pencilholders package. This is a different

// class, but one that shares the same simple name as the

// previous "CoffeeCup."

com.artima.pencilholders.CoffeeCup myPencilHolder =

new com.artima.pencilholders.CoffeeCup();

myCoffee.add(com.artima.vcafe.dishes.CoffeeCup.MAX\_SHORT\_ML);

myPencilHolder.add(10);

}

}

The code as shown above compiles fine, because each time you use a CoffeeCup you clearly indicate which CoffeeCup you want. You have indeed accomplished your goal of using two different CoffeeCup classes in the same source file, yet you have once again cluttered the code with long package names.

Fortunately, one other approach exists that may help you reduce some of the clutter. If you only import one of the packages containing a CoffeeCup class, you could use the simple name when referring to that CoffeeCup. As before, you'd have to use the fully qualified name when referring to the other CoffeeCup. Rewriting the previous example using this approach, yields the following code:

// In Source Packet in file packages/ex1/Example1e.java

// Import all types defined in com.artima.vcafe.dishes, but

// don't import anything from com.artima.pencilholders.

import com.artima.vcafe.dishes.\*;

class Example1e {

public static void main(String[] args) {

// Somewhere later in the code, you wish to instantiate a

// new CoffeeCup from the com.artima.vcafe.dishes package.

// Here you can just use the simple name:

CoffeeCup myCoffee = new CoffeeCup();

// To create a new CoffeeCup from the

// com.artima.pencilholders package, you must once again

// use the fully qualified name:

com.artima.pencilholders.CoffeeCup myPencilHolder =

new com.artima.pencilholders.CoffeeCup();

myCoffee.add(CoffeeCup.MAX\_TALL\_ML);

myPencilHolder.add(15);

}

}

You might be wondering if you can just import all the members of the com.artima package and just use vcafe.dishes.CoffeeCup and pencilholders.CoffeeCup to distinguish between the two classes of coffee cup. Well, you can't. The import statement only imports types, not packages. The statement "import com.artima.\*;" imports all the types defined in that package, but doesn't import any sub- packages defined in that package. The statement "import com.artima;" doesn't compile, because you are trying to import a package and not a class or interface. Another statement that doesn't compile is "import com.artima.\*.dishes;". The \* must always go at the end, as it only matches type names, not package names.

There is one exception to the rule that you must import types from other packages before you can use their simple names: java.lang.\*. The public types defined in the standard run- time library java.lang are automatically imported into every Java source file. This package contains classes, such as String, Thread, and Object, that are essential to the inner workings of Java programs. To make use of the types contained in the packages from Java's standard run-time library other than java.lang, you must either import the packages or use fully qualified names, just like any other package.

Import statements are provided as a convenience for the programmer. Because of import statements, you don't have to always type long and tedious fully qualified names. The Java compiler can work out the fully qualified names of types given the import statements and the simple names in a source file. When the compiler generates class files, it discards any import statements in the source file. In class files, all types are identified by their fully qualified names. In your programs, you can choose to use import statements or fully qualified names, whatever you think will maximize the readability of your code.

As mentioned earlier in this chapter, an import statement does not dynamically include code from a different file, as #include does for C and C++ programs. Import is just about names.

Access Levels for Types

One of the most useful features of Java packages is the ability to grant access to classes, interfaces, methods, or fields exclusively to other members of the same package. This feature gives the package an internal implementation and an external interface. It provides the usual advantages of a hidden internal implementation: robustness and ease of modification. The robustness arises from the inability of types declared in other packages to incorrectly manipulate the internal implementation of the package. Types in other packages must go through the external interface of the package, and the package maintains control of its internal implementation. Ease of modification comes from the ability to change the internal implementation of the package without affecting the code of other packages, which is tied only to the external interface.

The first step you can take to hide the internal implementation of a package is to declare as public only those types that are needed by other packages. When you declare a class or interface, it is by definition contained in a package. If you want a class or interface to be accessible to types declared in other packages, you must declare it public. If you do not declare it public, it will only be accessible to types in the same package. Therefore, you can denote some types (the public ones) as part of the external interface of the package. The other types (the ones that aren't public) are part of the internal implementation of the package. An example of both kinds of class declarations is shown below:

// In Source Packet in file

// packages/ex2/com/artima/vcafe/dishes/Cup.java

package com.artima.vcafe.dishes;

// Class Cup is part of the internal implementation of

// package com.artima.vcafe.dishes.

class Cup {

public void add(int amountOfCoffee) {

System.out.println("Adding to a Cup.");

}

//...

}

// In Source Packet in file

// packages/ex2/com/artima/vcafe/dishes/CoffeeCup.java

package com.artima.vcafe.dishes;

// Class CoffeeCup is part of the external interface of

// package com.artima.vcafe.dishes.

public class CoffeeCup extends Cup {

public static final int MAX\_SHORT\_ML = 237;

public static final int MAX\_TALL\_ML = 355;

public static final int MAX\_GRANDE\_ML = 473;

//...

}

In the code shown above, class CoffeeCup is declared public, but class Cup is not. Consequently, CoffeeCup is accessible everywhere, but Cup is accessible only in thecom.artima.vcafe.dishes package.

Package access is the default for types. Unless you explicitly modify your class declaration with the keyword public, you'll get "package access," as this default level of access is called. You cannot declare a class with the access specifiers protected or private. It must either be declared with the keyword public or have no access specifier.

As the example above demonstrates, you can declare a superclass with package access and still give its subclass public access. Given the code above, a type in another package could not subclass Cup, but could subclass CoffeeCup. If you want types in other packages to be able to use CoffeeCup but not subclass it, you must also declare it final, as shown below:

// In Source Packet in file

// packages/ex3/com/artima/vcafe/dishes/Cup.java

package com.artima.vcafe.dishes;

// Class Cup is part of the internal implementation of

// package com.artima.vcafe.dishes.

class Cup {

public void add(int amountOfCoffee) {

System.out.println("Adding to a Cup.");

}

//...

}

// In Source Packet in file

// packages/ex3/com/artima/vcafe/dishes/CoffeeCup.java

package com.artima.vcafe.dishes;

// Class CoffeeCup is part of the external interface of

// package com.artima.vcafe.dishes. It can be used, but

// not subclassed, by classes in other packages.

public final class CoffeeCup extends Cup {

public static final int MAX\_SHORT\_ML = 237;

public static final int MAX\_TALL\_ML = 355;

public static final int MAX\_GRANDE\_ML = 473;

//...

}

When you fill a package with types, you should separate the types that represent the implementation of the package from those that represent the interface. Only those types that are needed by other packages should be declared public. A good rule of thumb is to leave any class or interface with its default package access, unless you're sure it should be public.

Declaring a public class as final will prevent classes in other packages from declaring a subclass, but it will also restrict any other class in its own package from declaring a subclass. This is a severe restriction on the use of a class. Often you will want clients of your package to be able to subclass its public classes. That is one of the fundamental ways to reuse code in object-oriented programming. The rule of thumb here is to make classes final only when you have a good reason.

One possible reason to make a class final is to ensure your package will always behave as expected. For example, imagine you write a package that depends for correctness on the proper behavior of a certain class of objects, say the CoffeeCup class, defined in your package. You make class CoffeeCup public so that clients can create instances of it to pass to the methods of other classes defined in your package. If your package requires that the CoffeeCup objects passed to it behave in a certain way, your package might break if a client declares their own subclass of class CoffeeCup, say LeakyCup, and overrides the methods that your package depends upon for correctness. You can avoid this by declaring every method in CoffeeCup as final, or by declaring the entire CoffeeCup class final.

In the examples in this book, each type is declared in its own source file. The name of the source file is the name of the type plus the extension .java. For example, class CoffeeCup is declared in file CoffeeCup.java, and interface Washable is declared in file Washable.java. Although placing each type in a separate file named after the type is in general a good practice, because it makes the type's source easier for you and other developers to locate, it is not always required. Java compilers do require that public types be declared in a file that bears the name of the public types. They do not, however, require this of non-public types. You can place as many non-public types in the same file as you wish, and the file can have whatever name you wish. If a file does contain a public type, however, the file must be given the name of the public type. Because you can only have one package statement in each source file, all types declared in the same source file are members of the same package.

Access Levels for Class Members

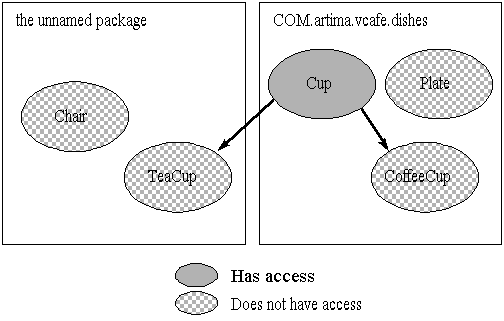
In general, within any class you design you will want to hide the implementation. Given that packages can (and should) be used to group related types, however, you may want to expose some fields and methods to other classes in the same package while keeping them hidden from classes outside the package. Java provides access control modifiers to support this intermediate level of implementation hiding. By applying proper modifiers on a class's fields and methods, you can hide the class's implementation from classes outside the package while exposing the implementation to classes inside the package.

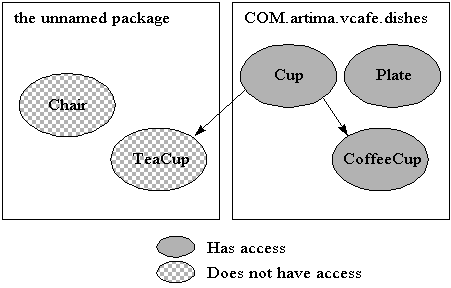
Java gives you three access control modifiers--private, protected, and public--to apply to the fields and methods of public classes, but you can obtain four distinct levels of access from their use. Three of the levels (private, protected, and public access) are denoted by the use of one of the three access control modifiers. The fourth level (package access) is the default and is indicated by the lack of any access control modifier. Here is a description of each of the four access levels available to members of public classes, in order from least to most accessible:

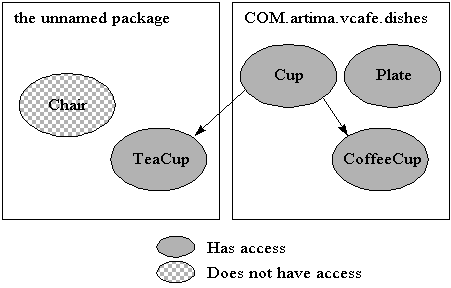
1. private access (denoted by the keyword private) - a field or method accessible only to the class which defines it.
2. package access (denoted by no access control modifier keywords) - a field or method accessible to any type in the same package.
3. protected access (denoted by the keyword protected) - a field or method accessible to any type in the same package, and to subclasses in any package.
4. public access (denoted by the keyword public) - a field or method accessible to any type in any package.

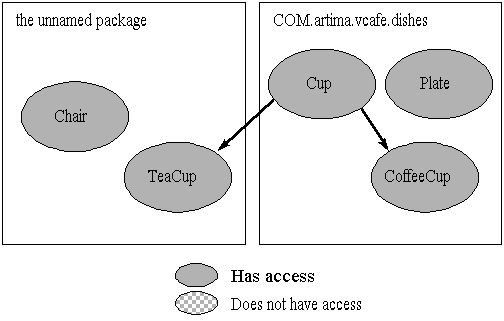
There is no way to grant special access to types in sub-packages. This is why the Java compiler and the Java Virtual Machine view a package and its sub-packages as independent packages with no special privileges between them. Thus, the relationship between types in hierarchically related packages, such as com.artima.vcafe and com.artima.vcafe.dishes, is only conceptual. Package hierarchies help you organize your types, but don't allow any special access privileges between the two groups of types.

A graphical depiction of the effect of each kind of access control modifier is shown in Figures 7-1 through 7-4. In these figures, the ovals represent classes, the arrows represent inheritance, the rectangles represent packages. Each figure indicates which classes will be able to access a member of class Cup with one of the five access levels. Classes that can access the member in Cupare shown in solid gray; classes that can't are shown with a checkerboard pattern.

  
Figure 5-1. Private access to a member of Cup.

  
Figure 5-2. Package access to a member of Cup.

  
Figure 5-3. Protected access to a member of Cup.

  
Figure 5-4. Public access to a member of Cup.

An example of each kind of access control modifier is shown in the following version of class Coffee:

// In Source Packet in file

// packages/ex4/com/artima/vcafe/beverages/Coffee.java

package com.artima.vcafe.beverages;

public class Coffee {

// PRIVATE ACCESS

// Accessible to only class Coffee itself.

private int temperature;

// PACKAGE ACCESS

// Accessible to Coffee and to the other classes and

// interfaces of package com.artima.vcafe.beverages.

void changeTemperature(int delta) {

temperature += delta;

}

// PROTECTED ACCESS

// Accessible to Coffee, to its subclasses (no matter what

// package the subclasses are defined in), and to the other

// types of package com.artima.vcafe.beverages, including

// non-subclasses.

protected static final int bestTemperature = 50;

// PUBLIC ACCESS

// Accessible to the entire universe.

public void setTemperature(int temperature) {

this.temperature = temperature;

}

public int getTemperature() {

return temperature;

}

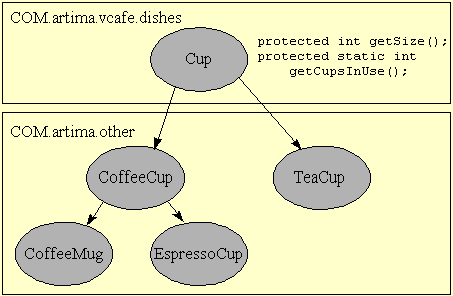
}

The True Meaning of private and protected

The private keyword grants exclusive access not to an object, but to a class. An object can access its private members, but so can any other object of the same class. For example, if aCoffeeCup object has a reference to another CoffeeCup object, the first CoffeeCup can access the second CoffeeCup's private members through that reference. This is true of both private variables and private methods, whether they are static or not.

Inside a package, the true meaning of the protected keyword is quite simple. To classes in the same package, protected access looks just like package access. Any class can access any protected member of another class declared in the same package.

When you have subclasses in other packages, however, the true meaning of protected becomes more complex. Take a look at the inheritance hierarchy shown in Figure 5-5. In this hierarchy, class Cup, which is declared in the com.artima.vcafe.dishes package, declares a protected instance method named getSize(). This method is accessible to any subclasses declared anywhere, including those shown declared in package com.artima.other. Any objects whose class descends from Cup--instances of class CoffeeCup, CoffeeMug, EspressoCup, or TeaCup-- can invoke getSize() on themselves. Whether they can invoke getSize() on a reference to another object, however, depends upon where that other object sits in the inheritance hierarchy.

  
Figure 5-5. The true meaning of protected.

If a protected instance variable or instance method is accessible to a class, that class can access the protected member through a reference only if the reference type is the class or one of its subclasses. For example, for code in the CoffeeCup class to invoke getSize() on a reference to another object, that reference must be of type CoffeeCup or one of its subclasses. ACoffeeCup object could therefore invoke getSize() on a CoffeeCup reference, a CoffeeMug reference, or an EspressoCup reference. A CoffeeCup object could not, however, invokegetSize() on a Cup reference or a TeaCup reference.

If class has a protected variable or method that is static, the rules are different. Take as an example the protected static method getCupsInUse() declared in class Cup as shown in Figure 5-5. Any code in a subclass of Cup can access a getCupsInUse() by invoking it on itself or invoking it on a reference of type Cup or any of its subclasses. Code in the EspressoCup class could invoke getCupsInUse() on itself or on a reference of type Cup, CoffeeCup, CoffeeMug, EspressoCup, or TeaCup.

Rules of Thumb for Class Member Access Levels

The most important rule of thumb concerning the use of access control modifiers is to keep data private unless you have a good reason not to. Keeping data private is the best way to maximize the robustness and ease of modification of your classes. If you keep data private, other classes can access a class's fields only through its methods. This enables the designer of a class to keep control over the manner in which the class's fields are manipulated. If fields are not private, other classes can change the fields directly, possibly in unpredictable and improper ways. Keeping data private also enables a class designer to more easily change the algorithms and data structures used by a class. Given that other classes can only manipulate a class's private fields indirectly, through the class's methods, other classes will depend only upon the external interface to the private fields provided by the methods. You can change the private fields of a class and modify the code of the methods that manipulate those fields. As long as you don't alter the signature and return type of the methods, the other classes that depended on the previous version of the class will still link properly. Making fields private is the fundamental technique for hiding the implementation of Java classes.

As mentioned in an earlier chapter, one other reason to make data private is because you synchronize access to data by multiple threads through methods. This justification for keeping data private will be discussed in Chapter 17.

As a general rule, the only good non-private field is a final one. Given that final fields cannot be changed after they are initialized, non-private final fields do not run the risk of improper manipulation by other classes. Other classes can use the field, but not change it.

A common use of non-private final fields is to define names to represent a set of valid values that may be passed to (or returned from) a method. As mentioned in Chapter 5, such fields are called constants and are declared static as well as non-private and final. A Java programmer will create constants in this manner in situations where a C++ programmer would have used an enumerated type or declared a "const" member variable.

Rules of thumb such as the ones outlined above are called rules of thumb for a reason: They are not absolute laws. Java allows you to declare fields in classes with any kind of access level, and you may very well encounter situations in which declaring a field private is too restrictive. One potential justification for non-private fields is simple trust. In some situations you may have absolute trust of certain other classes. For example, perhaps you are designing a small set of types that must work together closely to solve a particular problem. It may make sense to put all of these types in their own package, and allow them direct access to some of each other's fields. Although this would create interdependencies between the internal implementations of the classes, you may deem the level of interdependency to be acceptable. If later you change the internal implementation of one of the classes, you'll have to update the other classes that relied on the original implementation. As long as you don't grant access to the fields to classes outside the package, any repercussions of the implementation change will remain inside the package.

Nevertheless, the general rule of thumb in designing packages is to treat the types that share the same package with as much suspicion as types from different packages. If you don't trust classes from other packages to directly manipulate your class's fields, neither should you let classes from the same package directly manipulate them. Keep in mind that you usually can't prevent another programmer from adding new classes to your package, even if you only deliver class files to that programmer. If you leave all your fields with package access, a programmer using your package can easily gain access to those fields by creating a class and declaring it as a member of your package. Therefore, it is best to keep data private, except sometimes when the data is final, so that irrespective of what package classes are defined in, all classes must go through methods to manipulate each other's fields.

The methods you define in public classes should have whatever level of access control matches their role in your program. You should exploit the full range of access levels provided by Java on the methods of your public classes, assigning to each method the most restrictive access level it can reasonably have.

You can use the same rule of thumb to design classes that have package access. You must keep in mind, however, that for package-access classes, fields and methods declared public won't be accessible outside the package. Fields and methods declared protected won't be accessible to subclasses in other packages, because there won't be any subclasses in other packages. Only classes within the same package will be able to subclass the package-access class. Still, you should probably keep the same mindset when designing package-access classes as you do when designing public classes, because at some later time you may turn a package-access class into a public class.

Access Levels for Interfaces

Interfaces have slightly different rules for access levels, because every field and method defined by an interface is implicitly public. You can't use the keywords private or protected on the fields and methods of interfaces. If you leave off the public keyword when declaring interface members, as is officially recommended by the Java Language Specification, you do not get package access. You still get public access. Therefore, you can't hide any implementation details of a package inside an interface (You can't hide an interface's members). On the other hand, you can hide the entire interface. If you don't declare an interface public, the interface as a whole will only be available to other types in the same package. As with classes, you should make interfaces public only if they are needed by classes and interfaces defined in other packages.

Here's an example of two interfaces. Interface Soakable is part of the internal implementation of a package. Interface Washable is part of the external implementation of the package:

// In Source Packet in file

// packages/ex5/com/artima/vcafe/dishes/Washable.java

package com.artima.vcafe.dishes;

public interface Washable {

void wash();

}

// In Source Packet in file

// packages/ex5/com/artima/vcafe/dishes/Soakable.java

package com.artima.vcafe.dishes;

interface Soakable extends Washable {

void soak();

}

In this example, wash() and breakIt() are not explicitly declared public, because they are public by default. Because the Washable interface as a whole is not explicitly declared as public, however, it has package access. Interface Washable is only be accessible to other types declared in the com.artima.vcafe.dishes package. Interface Breakable, because it is declared as public, is available to any type declared in any package.

Mention This

The compiler gives default constructors the same access level as their class. In the example above, class CoffeeCup is public, so the default constructor is public. If CoffeeCup had been given package access (which will be defined in , the default constructor would be given package access as well.

Example: How Singleton pattern can be implemented using private constructors.