Thinking in Java

Fourth Edition

Bruce Eckel
President, MindView, Inc.

Access Control

Access control (or *implementation hiding*) is about "not getting it right the first time."

All good writers—including those who write software—know that a piece of work isn't good until it's been rewritten, often many times. If you leave a piece of code in a drawer for a while and come back to it, you may see a much better way to do it. This is one of the prime motivations for *refactoring*, which rewrites working code in order to make it more readable, understandable, and thus maintainable.[1]

There is a tension, however, in this desire to change and improve your code. There are often consumers (*client programmers*) who rely on some aspect of your code staying the same. So you want to change it; they want it to stay the same. Thus a primary consideration in object-oriented design is to "separate the things that change from the things that stay the same."

This is particularly important for libraries. Consumers of that library must rely on the part they use, and know that they won't need to rewrite code if a new version of the library comes out. On the flip side, the library creator must have the freedom to make modifications and improvements with the certainty that the client code won't be affected by those changes.

This can be achieved through convention. For example, the library programmer must agree not to remove existing methods when modifying a class in the library, since that would break the client programmer's code. The reverse situation is thornier, however. In the case of a field, how can the library creator know which fields have been accessed by client programmers? This is also true with methods that are only part of the implementation of a class, and not meant to be used directly by the client programmer. What if the library creator wants to rip out an old implementation and put in a new one? Changing any of those members might break a client programmer's code. Thus the library creator is in a strait jacket and can't change anything.

To solve this problem, Java provides *access specifiers* to allow the library creator to say what is available to the client programmer and what is not. The levels of access control from "most access" to "least access" are **public**, **protected**, package access (which has no keyword), and **private**. From the previous paragraph you might think that, as a library designer, you'll want to keep everything as "private" as possible, and expose only the methods that you want the client programmer to use. This is exactly right, even though it's often counterintuitive for people who program in other languages (especially C) and who are used to accessing everything without restriction. By the end of this chapter you should be convinced of the value of access control in Java.

The concept of a library of components and the control over who can access the components of that library is not complete, however. There's still the question of how the components are bundled together into a cohesive library unit. This is controlled with the **package** keyword in Java, and the access specifiers are affected by whether a class is in the same package or in a separate package. So to begin this chapter, you'll learn how library components are placed into packages. Then you'll be able to understand the complete meaning of the access specifiers.

---

[1] See *Refactoring: Improving the Design of Existing Code*, by Martin Fowler, et al. (Addison-Wesley, 1999). Occasionally someone will argue against refactoring, suggesting that code which works is perfectly good and it's a waste of time to refactor it. The problem with this way of thinking is that the lion's share of a project's time and money is not in the initial writing of the code, but in maintaining it. Making code easier to understand translates into very significant dollars.

Package caveat

It's worth remembering that anytime you create a package, you implicitly specify a directory structure when you give the package a name. The package *must* live in the directory indicated by its name, which must be a directory that is searchable starting from the CLASSPATH. Experimenting with the **package** keyword can be a bit frustrating at first, because unless you adhere to the package-name to directory-path rule, you'll get a lot of mysterious runtime messages about not being able to find a particular class, even if that class is sitting there in the same directory. If you get a message like this, try commenting out the **package** statement, and if it runs, you'll know where the problem lies.

Note that compiled code is often placed in a different directory than source code, but the path to the compiled code must still be found by the JVM using the CLASSPATH.

Java access specifiers

The Java access specifiers **public**, **protected**, and **private** are placed in front of each definition for each member in your class, whether it's a field or a method. Each access specifier only controls the access for that particular definition.

If you don't provide an access specifier, it means "package access." So one way or another, everything has some kind of access control. In the following sections, you'll learn about the various types of access.

Package access

All the examples before this chapter used no access specifiers. The default access has no keyword, but it is commonly referred to as *package access* (and sometimes "friendly"). It means that all the other classes in the current package have access to that member, but to all the classes outside of this package, the member appears to be **private**. Since a compilation unit—a file—can belong only to a single package, all the classes within a single compilation unit are automatically available to each other via package access.

Package access allows you to group related classes together in a package so that they can easily interact with each other. When you put classes together in a package, thus granting mutual access to their package-access members, you "own" the code in that package. It makes sense that only code that you own should have package access to other code that you own. You could say that package access gives a meaning or a reason for grouping classes together in a package. In many languages the way you organize your definitions in files can be arbitrary, but in Java you're compelled to organize them in a sensible fashion. In addition, you'll probably want to exclude classes that shouldn't have access to the classes being defined in the current package.

The class controls the code that has access to its members. Code from another package can't just come around and say, "Hi, I'm a friend of **Bob**'s!" and expect to be shown the **protected**, package-access, and **private** members of **Bob**. The only way to grant access to a member is to:

1. Make the member **public**. Then everybody, everywhere, can access it.

2. Give the member package access by leaving off any access specifier, and put the other classes in the same package. Then the other classes in that package can access the member.

3. As you'll see in the *Reusing Classes* chapter, when inheritance is introduced, an inherited class can access a **protected** member as well as a **public** member (but not **private** members). It can access package-access members only if the two classes are in the same package. But don't worry about inheritance and **protected** right now.

4. Provide "accessor/mutator" methods (also known as "get/set" methods) that read and change the value. This is the most civilized approach in terms of OOP, and it is fundamental to JavaBeans, as you'll see in the *Graphical User Interfaces* chapter.

public: interface access

When you use the **public** keyword, it means that the member declaration that immediately follows **public** is available to everyone, in particular to the client programmer who uses the library. Suppose you define a package **dessert** containing the following compilation unit:

```
//: access/dessert/Cookie.java
// Creates a library.
package access.dessert;

public class Cookie {
  public Cookie() {
   System.out.println("Cookie constructor");
  }
  void bite() { System.out.println("bite"); }
} ///:~
```

Remember, the class file produced by **Cookie.java** must reside in a subdirectory called **dessert**, in a directory under **access** (indicating the *Access Control* chapter of this book) that must be under one of the CLASSPATH directories. Don't make the mistake of thinking that Java will always look at the current directory as one of the starting points for searching. If you don't have a '**.**' as one of the paths in your CLASSPATH, Java won't look there.

Now if you create a program that uses **Cookie**:

```
//: access/Dinner.java
// Uses the library.
import access.dessert.*;

public class Dinner {
  public static void main(String[] args) {
    Cookie x = new Cookie();
    //! x.bite(); // Can't access
  }
} /* Output:
Cookie constructor
*///:~
```

you can create a **Cookie** object, since its constructor is **public** and the class is **public**. (We'll look more at the concept of a **public** class later.) However, the **bite( )** member is inaccessible inside **Dinner.java** since **bite( )** provides access only within package **dessert**, so the compiler prevents you from using it.

The default package

You might be surprised to discover that the following code compiles, even though it would appear that it breaks the rules:

```
//: access/Cake.java
// Accesses a class in a separate compilation unit.

class Cake {
  public static void main(String[] args) {
    Pie x = new Pie();
    x.f();
  }
} /* Output:
Pie.f()
*///:~
```

In a second file in the same directory:

```
//: access/Pie.java
// The other class.

class Pie {
  void f() { System.out.println("Pie.f()"); }
} ///:~
```

You might initially view these as completely foreign files, and yet **Cake** is able to create a **Pie** object and call its **f( )** method. (Note that you must have '.' in your CLASSPATH in order for the files to compile.) You'd typically think that **Pie** and **f( )** have package access and are therefore not available to **Cake**. They *do* have package access—that part is correct. The reason that they are available in **Cake.java** is because they are in the same directory and have no explicit package name. Java treats files like this as implicitly part of the "default package" for that directory, and thus they provide package access to all the other files in that directory.

private: you can't touch that!

The **private** keyword means that no one can access that member except the class that contains that member, inside methods of that class. Other classes in the same package cannot access **private** members, so it's as if you're even insulating the class against yourself. On the other hand, it's not unlikely that a package might be created by several people collaborating together, so **private** allows you to freely change that member without concern that it will affect another class in the same package.

The default package access often provides an adequate amount of hiding; remember, a packageaccess member is inaccessible to the client programmer using the class. This is nice, since the default access is the one that you normally use (and the one that you'll get if you forget to add any access control). Thus, you'll typically think about access for the members that you explicitly want to make **public** for the client programmer, and as a result, you might initially think that you won't use the **private** keyword very often, since it's tolerable to get away without it. However, it turns out that the consistent use of **private** is very important, especially where multithreading is concerned. (As you'll see in the *Concurrency* chapter.)

Here's an example of the use of **private**:

```
//: access/IceCream.java
// Demonstrates "private" keyword.
```

```
class Sundae {
  private Sundae() {}
  static Sundae makeASundae() {
    return new Sundae();
  }
}

public class IceCream {
  public static void main(String[] args) {
    //! Sundae x = new Sundae();
    Sundae x = Sundae.makeASundae();
  }
} ///:~
```

This shows an example in which **private** comes in handy: You might want to control how an object is created and prevent someone from directly accessing a particular constructor (or all of them). In the preceding example, you cannot create a **Sundae** object via its constructor; instead, you must call the **makeASundae( )** method to do it for you.[4]

Any method that you're certain is only a "helper" method for that class can be made **private**, to ensure that you don't accidentally use it elsewhere in the package and thus prohibit yourself from changing or removing the method. Making a method **private** guarantees that you retain this option.

The same is true for a **private** field inside a class. Unless you must expose the underlying implementation (which is less likely than you might think), you should make all fields **private**. However, just because a reference to an object is **private** inside a class doesn't mean that some other object can't have a **public** reference to the same object. (See the online supplements for this book to learn about aliasing issues.)

protected: inheritance access

Understanding the **protected** access specifier requires a jump ahead. First, you should be aware that you don't need to understand this section to continue through this book up through inheritance (the *Reusing Classes* chapter). But for completeness, here is a brief description and example using **protected**.

The **protected** keyword deals with a concept called *inheritance*, which takes an existing class— which we refer to as the *base class*—and adds new members to that class without touching the existing class. You can also change the behavior of existing members of the class. To inherit from a class, you say that your new class **extends** an existing class, like this:

```
class Foo extends Bar {
```

The rest of the class definition looks the same.

If you create a new package and inherit from a class in another package, the only members you have access to are the **public** members of the original package. (Of course, if you perform the inheritance in the *same* package, you can manipulate all the members that have package access.) Sometimes the creator of the base class would like to take a particular member and grant access to derived classes but not the world in general. That's what **protected** does. **protected** also gives package access—that is, other classes in the same package may access **protected** elements.

---

[4] There's another effect in this case: Since the default constructor is the only one defined, and it's **private**, it will prevent inheritance of this class. (A subject that will be introduced later.)

If you refer back to the file **Cookie.java**, the following class *cannot* call the package-access member **bite( )**:

```
//: access/ChocolateChip.java
// Can't use package-access member from another package.
import access.dessert.*;

public class ChocolateChip extends Cookie {
  public ChocolateChip() {
   System.out.println("ChocolateChip constructor");
  }
  public void chomp() {
    //! bite(); // Can't access bite
  }
  public static void main(String[] args) {
    ChocolateChip x = new ChocolateChip();
    x.chomp();
  }
} /* Output:
Cookie constructor
ChocolateChip constructor
*///:~
```

One of the interesting things about inheritance is that if a method **bite( )** exists in class **Cookie**, then it also exists in any class inherited from **Cookie**. But since **bite( )** has package access and is in a foreign package, it's unavailable to us in this one. Of course, you could make it **public**, but then everyone would have access, and maybe that's not what you want. If you change the class **Cookie** as follows:

```
//: access/cookie2/Cookie.java
package access.cookie2;

public class Cookie {
  public Cookie() {
    System.out.println("Cookie constructor");
  }
  protected void bite() {
    System.out.println("bite");
  }
} ///:~
```

now **bite( )** becomes accessible to anyone inheriting from **Cookie**:

```
//: access/ChocolateChip2.java
import access.cookie2.*;

public class ChocolateChip2 extends Cookie {
  public ChocolateChip2() {
   System.out.println("ChocolateChip2 constructor");
  }
  public void chomp() { bite(); } // Protected method
  public static void main(String[] args) {
    ChocolateChip2 x = new ChocolateChip2();
    x.chomp();
  }
} /* Output:
Cookie constructor
ChocolateChip2 constructor
bite
*///:~
```

Note that, although **bite( )** also has package access, it is *not* **public**.

Exercise 4: (2) Show that **protected** methods have package access but are not **public**.

Exercise 5: (2) Create a class with **public**, **private**, **protected**, and package-access fields and method members. Create an object of this class and see what kind of compiler messages you get when you try to access all the class members. Be aware that classes in the same directory are part of the "default" package.

Exercise 6: (1) Create a class with **protected** data. Create a second class in the same file with a method that manipulates the **protected** data in the first class.

Interface and implementation

Access control is often referred to as *implementation hiding*. Wrapping data and methods within classes in combination with implementation hiding is often called *encapsulation*.[5] The result is a data type with characteristics and behaviors.

Access control puts boundaries within a data type for two important reasons. The first is to establish what the client programmers can and can't use. You can build your internal mechanisms into the structure without worrying that the client programmers will accidentally treat the internals as part of the interface that they should be using.

This feeds directly into the second reason, which is to separate the interface from the implementation. If the structure is used in a set of programs, but client programmers can't do anything but send messages to the **public** interface, then you are free to change anything that's *not* **public** (e.g., package access, **protected**, or **private**) without breaking client code.

For clarity, you might prefer a style of creating classes that puts the **public** members at the beginning, followed by the **protected**, package-access, and **private** members. The advantage is that the user of the class can then read down from the top and see first what's important to them (the **public** members, because they can be accessed outside the file), and stop reading when they encounter the non-**public** members, which are part of the internal implementation:

```
//: access/OrganizedByAccess.java

public class OrganizedByAccess {
  public void pub1() { /* ... */ }
  public void pub2() { /* ... */ }
  public void pub3() { /* ... */ }
  private void priv1() { /* ... */ }
  private void priv2() { /* ... */ }
  private void priv3() { /* ... */ }
  private int i;
  // ...
} ///:~
```

This will make it only partially easier to read, because the interface and implementation are still mixed together. That is, you still see the source code—the implementation—because it's right there in the class. In addition, the comment documentation supported by Javadoc lessens the importance of code readability by the client programmer. Displaying the interface to the consumer of a class is really the job of the *class browser*, a tool whose job is to look at all the available classes and show you what you can do with them (i.e., what members are

[5] However, people often refer to implementation hiding alone as encapsulation.

Summary

In any relationship it's important to have boundaries that are respected by all parties involved. When you create a library, you establish a relationship with the user of that library—the client programmer—who is another programmer, but one using your library to build an application or a bigger library.

Without rules, client programmers can do anything they want with all the members of a class, even if you might prefer they don't directly manipulate some of the members. Everything's naked to the world.

This chapter looked at how classes are built to form libraries: first, the way a group of classes is packaged within a library, and second, the way the class controls access to its members.

It is estimated that a C programming project begins to break down somewhere between 50K and 100K lines of code because C has a single namespace, and names begin to collide, causing extra management overhead. In Java, the **package** keyword, the package naming scheme, and the **import** keyword give you complete control over names, so the issue of name collision is easily avoided.

There are two reasons for controlling access to members. The first is to keep users' hands off portions that they shouldn't touch. These pieces are necessary for the internal operations of the class, but not part of the interface that the client programmer needs. So making methods and fields **private** is a service to client programmers, because they can easily see what's important to them and what they can ignore. It simplifies their understanding of the class.

The second and most important reason for access control is to allow the library designer to change the internal workings of the class without worrying about how it will affect the client programmer. You might, for example, build a class one way at first, and then discover that restructuring your code will provide much greater speed. If the interface and implementation are clearly separated and protected, you can accomplish this without forcing client programmers to rewrite their code. Access control ensures that no client programmer becomes dependent on any part of the underlying implementation of a class.

When you have the ability to change the underlying implementation, you not only have the freedom to improve your design, you also have the freedom to make mistakes. No matter how carefully you plan and design, you'll make mistakes. Knowing that it's relatively safe to make these mistakes means you'll be more experimental, you'll learn more quickly, and you'll finish your project sooner.

The public interface to a class is what the user *does* see, so that is the most important part of the class to get "right" during analysis and design. Even that allows you some leeway for change. If you don't get the interface right the first time, you can *add* more methods, as long as you don't remove any that client programmers have already used in their code.

Notice that access control focuses on a relationship—and a kind of communication—between a library creator and the external clients of that library. There are many situations where this is not the case. For example, you are writing all the code yourself, or you are working in close quarters with a small team and everything goes into the same package. These situations have a different kind of communication, and rigid adherence to access rules may not be optimal. Default (package) access may be just fine.