IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 15, Issue 1 (Sep. - Oct. 2013), PP 59-66
www.iosrjournals.org

"Study of Java Access Control Mechanism"

Manhal Mohammad Basher
(Department Of Mechanical, Technical Institute / Foundation of Technical Education, Iraq)

Abstract: *a class as "a collection of data and methods." One of the important object-oriented techniques is hiding the data within the class and making it available only through the methods. This technique is known as encapsulation because it seals the data (and internal methods) safely inside the "capsule" of the class, where it can be accessed only by trusted users.*

Another reason for encapsulation is to protect your class against accidental or willful stupidity. A class often contains a number of interdependent fields that must be in a consistent state. If you allow a users to manipulate those fields directly, he may change one field without changing important related fields, thus leaving the class in an inconsistent state. If, instead, he has to call a method to change the field, that method can be sure to do everything necessary to keep the state consistent. Similarly, if a class defines certain methods for internal use only, hiding these methods prevents users of the class from calling them.

The goal is : *All the fields and methods of a class can always be used within the body of the class itself. Java defines access control rules that restrict members of a class from being used outside the class that is done through keeping the data (and internal methods) safely inside the "capsule" of the class, where it can be accessed only by trusted users.*

Keywords: *Encapsulation, Access Control, Classes, Constructors, Methods.*

I. Introduction

Every time you download a "normal" program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer's local file system. In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack.

Java supplies a rich set of access specifiers. Some aspects of access control are related mostly to encapsulation, inheritance and packages. Also Java's access specifiers are public, private, and protected. Java also defines a default access level. protected applies only when inheritance is involved.

Encapsulation links data with the code that manipulates it. However, encapsulation provides important attribute: access control. Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. Also it is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding.

II. Theory Part

2.1 Object-Oriented Programming

Object-oriented programming (OOP) is at the core of Java. In fact, all Java programs are to at least some extent object-oriented. OOP is so integral to Java that it is best to understand its basic principles before you begin writing even simple Java programs.

There are Four OOP Principles

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are abstraction, encapsulation, inheritance, and polymorphism.

We are talking about encapsulation and inheritance because these two things is related with Access Control that is related with computer network and internet.

2.1.1 Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. A class defines the

structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as instances of a class. Thus, a class is a logical construct; an object has physical reality. When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called members of the class. Specifically, the data defined by the class are referred to as member variables or instance variables. The code that operates on that data is referred to as member methods or just methods. In properly written Java programs, the methods define how the member variables can be used. This means that the behavior and interface of a class are defined by the methods that operate on its instance data. Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked private or public. The public interface of a class represents everything that external users of the class need to know, or may know. The private methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable. Since the private members of a class may only be accessed by other parts of your program through the class' public methods, you can ensure that no improper actions take place. Of course, this means that the public interface should be carefully designed not to expose too much of the inner workings of a class.[1]

2.1.2 Inheritance

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. As mentioned earlier, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification dog, which in turn is part of the mammal class, which is under the larger class animal. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes plus any that it adds as part of its specialization. This is a key concept that lets object-oriented programs grow in complexity linearly rather than geometrically. A new subclass inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.[2]

III. Practical Part

3.1 Access Control Mechanism With Encapsulation

Encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: access control. Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. For example, allowing access to data only through a well defined set of methods, you can prevent the misuse of that data., you will be introduced to the mechanism by which you can precisely control access to the various members of a class. How a member can be accessed is determined by the access specifier that modifies its declaration. Java supplies a rich set of access specifiers. Some aspects of access control are related mostly to inheritance or packages. (A package is, essentially, a grouping of classes.). Here, let's begin by examining access control as it applies to a single class. Once you understand the fundamentals of access control, the rest will be easy.

Java's access specifiers are public, private, and protected. Java also defines a default access level. protected applies only when inheritance is involved. The other access specifiers are described next. Let's begin by defining public and private. When a member of a class is modified by the public specifier, then that member can be accessed by any other code. When a member of a class is specified as private, then that member can only be accessed by other members of its class. Now you can understand why main( ) has always been preceded by the public specifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. In the classes developed so far, all members of a class have used the default access mode, which is essentially public. However, this is not what you will typically want to be the case. Usually, you will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when you will want to define methods that are private to a class.

An access specifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here is an example:

```
public int i;
private double j;
private int myMethod(int a, char b)
```

```
{
/* This program demonstrates the difference between
public and private.
*/
class Test
    {
        int a;                  // default access
        public int b;           // public access
        private int c;          // private access

        // methods to access c
        void setc(int i)
        {
        // set c's value
        c = i;
        }

        int getc()
        {               // get c's value
        return c;
        }
    }
class AccessTest
        {
                public static void main(String args[])
                {
                Test ob = new Test();
                // These are OK, a and b may be accessed directly
                ob.a = 10;
                ob.b = 20;
                // This is not OK and will cause an error
                // ob.c = 100;                          // Error!
                // You must access c through its methods
                ob.setc(100);                           // OK
                System.out.println("a, b, and c: " + ob.a + " " +
                ob.b + " " + ob.getc());
                }
        }
```

As you can see, inside the Test class, a uses default access, which for this example is the same as specifying public. b is explicitly specified as public. Member c is given private access. This means that it cannot be accessed by code outside of its class. So, inside the AccessTest class, c cannot be used directly. It must be accessed through its public methods: setc( ) and getc( ). If you were to remove the comment symbol from the beginning of the following line,

// ob.c = 100; // Error!

then you would not be able to compile this program because of the access violation.[3]

3.2 Access Control Mechanism With Inheritance

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a superclass. The class that does the inheriting is called a subclass. Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as private. For example, consider the following simple class hierarchy:

```
/*
```

In a class hierarchy, private members remain private to their class. This program contains an error and will not compile.

```
*/
// Create a superclass.
class A
{
int i;                          // public by default
private int j;                  // private to A
        void setij(int x, int y)
        {
        i = x;
        j = y;
        }
}
// A's j is not accessible here.
class B extends A
{
int total;
        void sum()
        {
        total = i + j;                  // ERROR, j is not accessible here
        }
}

class Access
{
        public static void main(String args[])
        {
        B subOb = new B();
        subOb.setij(10, 12);
        subOb.sum();
        System.out.println("Total is " + subOb.total);
       }
}
```

This program will not compile because the reference to j inside the sum( ) method of B causes an access violation. Since j is declared as private, it is only accessible by other embers of its own class. Subclasses have no access to it.

REMEMBER A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

3.3 Access Control Mechanism With Packages

Packages are containers for classes that are used to keep the class name space compartmentalized. For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

Defining a Package

To create a package is quite easy: simply include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name. While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code. This is the general form of the package statement:

package pkg;

Here, pkg is the name of the package. For example, the following statement creates a package called MyPackage.

package MyPackage;

Java uses file system directories to store packages. For example, the .class files for any classes you declare to be part of MyPackage must be stored in a directory called MyPackage. Remember that case is significant, and the

directory name must match the package name exactly. 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.

3.3.1 Access Protection

Packages add another dimension to access control. Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages. Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access specifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories. While Java's access control mechanism may seem complicated, we can simplify it as follows. Anything declared public can be accessed from anywhere. Anything declared private cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.

An Access Example

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. Remember that the classes for the two different packages need to be stored in directories named after their respective packages in this case, p1 and p2. The source for the first package defines three classes: Protection, Derived, and Same Package. The first class defines four int variables in each of the legal protection modes. The variable n is declared with the default protection, n_pri is private, n_pro is protected, and n_pub is public.

Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out. Before each of these lines is a comment listing the places from which this level of protection would allow access. The second class, Derived, is a subclass of Protection in the same package, p1. This grants Derived access to every variable in Protection except for n_pri, the private one. The third class, SamePackage, is not a subclass of Protection, but is in the same package and also has access to all but n_pri. Private No Modifier Protected Public

This is file Protection.java:

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

This is file Derived.java:

```
package p1;
class Derived extends Protection
```

```
{
        Derived()
        {
        System.out.println("derived constructor");
        System.out.println("n = " + n);
        // class only
        // System.out.println("n_pri = "4 + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
        }
}
```

This is file SamePackage.java:

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

Following is the source code for the other package, p2. The two classes defined in p2 cover the other two conditions that are affected by access control. The first class, Protection2, is a subclass of p1.Protection. This grants access to all of p1.Protection's variables except for n_pri (because it is private) and n, the variable declared with the default protection. emember, the default only allows access from within the class or the package, not extra-package subclasses. Finally, the class OtherPackage has access to only one variable, n_pub, which was declared public.[4] [5]

This is file Protection2.java:

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

This is file OtherPackage.java:

```
package p2;
class OtherPackage
{
        OtherPackage()
        {
        p1.Protection p = new p1.Protection();
        System.out.println("other package constructor");
        // class or package only
```

```
        // System.out.println("n = " + p.n);
        // class only
        // System.out.println("n_pri = " + p.n_pri);
        // class, subclass or package only
        // System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
        }
}
```

If you wish to try these two packages, here are two test files you can use. The one for package p1 is shown here:

```
// Demo package p1.
package p1;
// Instantiate the various classes in p1.
public class Demo
{
        public static void main(String args[])
        {
        Protection ob1 = new Protection();
        Derived ob2 = new Derived();
        SamePackage ob3 = new SamePackage();
        }
}
```

The test file for p2 is shown next:

```
// Demo package p2.
package p2;
// Instantiate the various classes in p2.
public class Demo
{
        public static void main(String args[])
        {
        Protection2 ob1 = new Protection2();
        OtherPackage ob2 = new OtherPackage();
        }
}
```

IV. Conclusion

In Java Access Specifiers The access to classes, constructors, methods and fields are regulated using access modifiers i.e. a class can control what information or data can be accessible by other classes. To take advantage of encapsulation, you should minimize access whenever possible.

Java provides a number of access modifiers to help you set the level of access you want for classes as well as the fields, methods and constructors in your classes. A member has package or default accessibility when no accessibility modifier is specified.

Access Modifiers

1. private
2. protected
3. default
4. public

public access modifier

Fields, methods and constructors declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package.

private access modifier

The private (most restrictive) fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods or constructors declared private are strictly controlled, which means they cannot be accesses by anywhere outside the enclosing class. A standard design strategy is to make all fields private and provide public getter methods for them.

protected access modifier

The protected fields or methods cannot be used for classes and Interfaces. It also cannot be used for fields and methods within an interface. Fields, methods and constructors declared protected in a superclass can be accessed only by subclasses in other packages. Classes in the same package can also access protected fields, methods and constructors as well, even if they are not a subclass of the protected member's class.

Default access modifier

Java provides a default specifier which is used when no access modifier is present. Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package. The default modifier is not used for fields and methods within an interface.

Reference

[1] L. Gong and G. Ellison, Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation. Pearson Education, 2003.

[2] L. Gong, M. Mueller, H. Prafullchandra, R. Schemers, and S. Microsystems, "Going beyond the sandbox: An overview of the new security architecture in the Java development kit 1.2," Proceedings of the USENIX Symposium for Secure Systems, 1997.

[3] N. Hardy, "The confused deputy," SIGOPS Operating Systems Review, vol. 22, no. 4, pp. 36–38, 1988.

[4] I. Welch and R. Stroud, "Supporting real world security models in Java," in Distributed Computing Systems, 1999. Proceedings. 7th IEEE Workshop on Future Trends of Distributed Computing Systems, 1999, pp. 155–159.

[5] J. Viega, J. Bloch, and P. Chandra, "Applying Aspect-Oriented programming to security," Cutter IT Journal, vol. 14, no. 2, pp. 31–39, Feb. 2001.