**Java Access Modifiers**

A *Java access modifier* specifies which classes can access a given class and its fields, constructors and methods. Access modifiers can be specified separately for a class, its constructors, fields and methods. Java access modifiers are also sometimes referred to in daily speech as *Java access specifiers*, but the correct name is Java access modifiers. Classes, fields, constructors and methods can have one of four different Java access modifiers:

* private
* default (package)
* protected
* public

Each of these Java access modifiers will be covered in the following sections of this Java access modifier tutorial. The following table summarizes what Java constructs each Java access modifier can be applied to:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **private** | **default** | **protected** | **public** |
| Class | No | Yes | No | Yes |
| Nested Class | Yes | Yes | Yes | Yes |
| Constructor | Yes | Yes | Yes | Yes |
| Method | Yes | Yes | Yes | Yes |
| Field | Yes | Yes | Yes | Yes |

Assigning an access modifier to a class, constructor, field or method is also sometimes referred to as "marking" that class, constructor, field or method as that which the access modifier specifies. For instance, assigning the Java access modifier public to a method would be referred to as marking the method as public.

**private Access Modifier**

If a method or variable is marked as private (has the private access modifier assigned to it), then only code inside the same class can access the variable, or call the method. Code inside subclasses cannot access the variable or method, nor can code from any external class.

Classes cannot be marked with the private access modifier. Marking a class with the private access modifier would mean that no other class could access it, which means that you could not really use the class at all. Therefore the private access modifier is not allowed for classes.

Here is an example of assigning the private access modifier to a field:

public class Clock {

**private** long time = 0;

}

The member variable time has been marked as private. That means, that the member variable timeinside the Clock class cannot be accessed from code outside the Clock class.

**Accessing private Fields via Accessor Methods**

Fields are often declared private to control the access to them from the outside world. In some cases the fields are truly private, meaning they are only used internally in the class. In other cases the fields can be accessed via accessor methods (e.g. getters and setters). Here is an accessor method example:

public class Clock {

private long time = 0;

public long getTime() {

return this.time;

}

public void setTime(long theTime) {

this.time = theTime;

}

}

In the above example the two methods getTime() and setTime() can access the time member variable. The two methods are declared public, meaning they can be called from code anywhere in your application. The public Java access modifier is covered later in this text.

**private Constructors**

If a constructor in a class is assigned the private Java access modifier, that means that the constructor cannot be called from anywhere outside the class. A private constructor can still get called from other constructors, or from static methods in the same class. Here is a Java class example illustrating that:

public class Clock {

private long time = 0;

private Clock(long time) {

this.time = time;

}

public Clock(long time, long timeOffset) {

this(time);

this.time += timeOffset;

}

public static Clock newClock() {

return new Clock(System.currentTimeMillis());

}

}

This version of the Clock class contains a private constructor and a public constructor. The privateconstructor is called from the public constructor (the statement this();). The private constructor is also called from the static method newClock().

The above example only serves to show you that a private constructor can be called from public constructors and from static methods inside the same class. Do not perceive the above example as an example of clever design in any way.

**default (package) Access Modifier**

The default Java access modifier is declared by not writing any access modifier at all. The default access modifier means that code inside the class itself as well as code inside classes in the same package as this class, can access the class, field, constructor or method which the default access modifier is assigned to. Therefore, the default access modifier is also sometimes referred to as the package access modifier. Subclasses cannot access methods and member variables (fields) in the superclass, if they these methods and fields are marked with the default access modifier, unless the subclass is located in the same package as the superclass.

Here is an default / package access modifier example:

public class Clock {

long time = 0;

}

public class ClockReader {

Clock clock = new Clock();

public long readClock{

return clock.time;

}

}

The time field in the Clock class has no access modifier, which means that it is implicitly assigned the default / package access modifier. Therefore, the ClockReader class can read the time member variable of the Clock object, provided that ClockReader and Clock are located in the same Java package.

**protected Access Modifier**

The protected access modifier provides the same access as the default access modifier, with the addition that subclasses can access protected methods and member variables (fields) of the superclass. This is true even if the subclass is not located in the same package as the superclass.

Here is a protected access modifier example:

public class Clock {

**protected** long time = 0; // time in milliseconds

}

public class SmartClock() extends Clock{

public long getTimeInSeconds() {

return this.time / 1000;

}

}

In the above example the subclass SmartClock has a method called getTimeInSeconds() which accesses the time variable of the superclass Clock. This is possible even if Clock and SmartClock are not located in the same package, because the time field is marked with the protected Java access modifier.

**public Access Modifier**

The Java access modifier public means that all code can access the class, field, constructor or method, regardless of where the accessing code is located. The accessing code can be in a different class and different package.

Here is a public access modifier example:

**public** class Clock {

**public** long time = 0;

}

public class ClockReader {

Clock clock = new Clock();

public long readClock{

return clock.time;

}

}

The time field in the Clock class is marked with the public Java access modifier. Therefore, the ClockReader class can access the time field in the Clock no matter what package the ClockReader is located in.

**Class Access Modifiers**

It is important to keep in mind that the Java access modifier assigned to a Java class takes precedence over any access modifiers assigned to fields, constructors and methods of that class. If the class is marked with the default access modifier, then no other class outside the same Java package can access that class, including its constructors, fields and methods. It doesn't help that you declare these fields public, or even public static.

The Java access modifiers **private and protected cannot be assigned to a class**. Only to constructors, methods and fields inside classes. Classes can only have the default (package) and public access modifier assigned to them.

**Interface Access Modifiers**

Java interfaces are meant to specify fields and methods that are publicly available in classes that implement the interfaces. Therefore you cannot use the private and protected access modifiers in interfaces. Fields and methods in interfaces are implicitly declared public if you leave out an access modifier, so you cannot use the default access modifier either (no access modifier).

**Access Modifiers and Inheritance**

When you create a subclass of some class, the methods in the subclass cannot have less accessible access modifiers assigned to them than they had in the superclass. For instance, if a method in the superclass is public then it must be public in the subclass too, in case the subclass overrides the method. If a method in the superclass is protected then it must be either protected or public in the subclass.

While it is not allowed to decrease accessibility of an overridden method, it is allowed to expand accessibility of an overridden method. For instance, if a method is assigned the default access modifier in the superclass, then it is allowed to assign the overridden method in the subclass the public access modifier.