Modifiers

Java modifiers are divided into two groups:

- **Access Modifiers** - controls the access level
- **Non-Access Modifiers -** do not control access level, but provides other functionality

Access Modifiers

An access modifier which is used to set the access level/visibility for **classes**, **fields**, **methods** and **properties**.

For **classes**, you can use either public or *default*:

Modifier	Description
public	The class is accessible by any other class
default	The class is only accessible by classes in the same package. This is used when you don't specify a modifier.

For **attributes, methods and constructors**, you can use the one of the following:

Modifier	Description
public	The code is accessible for all classes
private	The code is only accessible within the declared class
default	The code is only accessible in the same package. This is used when you don't specify a modifier.
protected	The code is accessible in the same package and **subclasses**.

Private Modifier

If you declare a field with a private access modifier, it can only be accessed within the same class:

Example

```
class Clothes
{
   private String color = "pink"; // The modifier is private
```

```
    public static void main(String[] args)
    {
        Clothes myClothes = new Clothes();
        System.out.println(myClothes.color);
    }
}
```

Public Modifier

If you declare a field with a public access modifier, it is accessible for all classes:

Example

```
class Clothes
{
    public String color = "pink"; // The modifier is public
}

class Program
{
    public static void main(String[] args)
    {
        Clothes myClothes = new Clothes();
        // The following statement will cause an error while the modifier of color is private
        System.out.println(myClothes.color);
    }
}
```

Note: By default, all members of a class are private if you don't specify an access modifier:

Example

```
class Clothes
{
    String size; // The default modifier is private
    String color; // The default modifier is private
}
```

Encapsulation

The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

declare fields/variables as private

provide public get and set methods to access and update the value of a private field

Get and Set

You learned from the previous chapter that private variables can only be accessed within the same class (an outside class has no access to it). However, it is possible to access them if we provide public get and set methods.

The get method returns the variable value, and the set method sets the value.

Syntax

The get and set methods start with either get or set, followed by the name of the variable, with the first letter in upper case:

Example

```
public class Clothes {
   private String color; // private = restricted access

   public String getColor() { // get method
      return color;
   }

   public void setColor(String color) { // set method
      this.color = color;
   }
}
```

Explanation

The Color property is associated with the color field, and the property name with an uppercase first letter.

The **get method** returns the value of the variable color. The **set method** assigns a value to the color variable.

Now we can use the Color property to access and update the private field of the Clothes class:

Example

```
class Clothes
{
   protected String color; // field
   public String getColor() // get method
   {
      return color;
   }

   public void setColor(String color) { // set method
      this.color = color;
   }
}

class Program
{
   public static void main(String[] args)
   {
      Clothes myClothes = new Clothes();
      myClothes.color = "yellow";
      System.out.println(myClothes.getColor());
   }
}
```

Non-Access Modifiers

For **classes**, you can use either final or abstract:

Modifier	**Description**
final	The class cannot be inherited by other classes.
abstract	The class cannot be used to create objects.

For **attributes and methods**, you can use the one of the following:

Modifier	**Description**
final	Attributes and methods cannot be overridden/modified
static	Attributes and methods belongs to the class, rather than an object
abstract	Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example **abstract void run();**. The body is provided by the subclass (inherited from).
transient	Attributes and methods are skipped when serializing the object containing them
synchronized	Methods can only be accessed by one thread at a time
volatile	The value of an attribute is not cached thread-locally, and is always read from the "main memory"

Example

```
private static final double[] PROBABILITIES = { 0.025, 0.025, 0.05, 0.05, 0.05, 0.30, 0.50 };

private static final int GUARANTEED_DRAW = 80;
```

Example

```
class FinalClass {
  public final void finalMethod() // create a final method
  {
    System.out.println("This is a final method.");
  }
}

class Program extends FinalClass {
  public final void finalMethod() { // try to override final method
    System.out.println("The final method is overridden.");
  }

  public static void main(String[] args) {
    Program program = new Program();
    program.finalMethod();
  }
}
```

Explanation

The above example will cause a compilation error because the final methods cannot be overridden/modified.