Access Modifier

OODP, 2022
w3schools

Java Modifier

We divide modifiers into two groups:

Access Modifiers - controls the access level

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

```
final class Vehicle {
  protected String brand = "Ford";
  public void honk() {
    System.out.println("Tuut, tuut!");
  }
}
```

Try compile...

```
class VehicleMain extends Vehicle { //Error
  private String modelName = "Mustang";
  public static void main(String[] args) {
    VehicleMain myFastCar = new VehicleMain();
    myFastCar.honk();
    System.out.println(myFastCar.brand + " " + myFastCar.modelName);
  }
}
```

W3Schools

Access Modifiers

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. You will learn more about packages in the Packages chapter

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

Modifier	Description	Try it
`public`	The code is accessible for all classes	Try it »
`private`	The code is only accessible within the declared class	Try it »
default	The code is only accessible in the same package. This is used when you don't specify a modifier. You will learn more about packages in the Packages chapter	Try it »
`protected`	The code is accessible in the same package and **subclasses**. You will learn more about subclasses and superclasses in the Inheritance chapter	Try it »

```
class DefaultVehicle {
  protected String brand = "Ford";
  int plateNumber = 123;
  public void honk() {
    System.out.println("Tuut, tuut!");
  }
}
```

Fix the code to be compiled.

```
class VehicleMain extends DefaultVehicle {
  private String modelName = "Mustang";
  public static void main(String[] args) {
    VehicleMain myFastCar = new VehicleMain();
    myFastCar.honk();
    System.out.println(myFastCar.brand + " " + myFastCar.modelName);
    System.out.println( "Plate Number " + plateNumber);
  }
}
```

Non-Access Modifiers

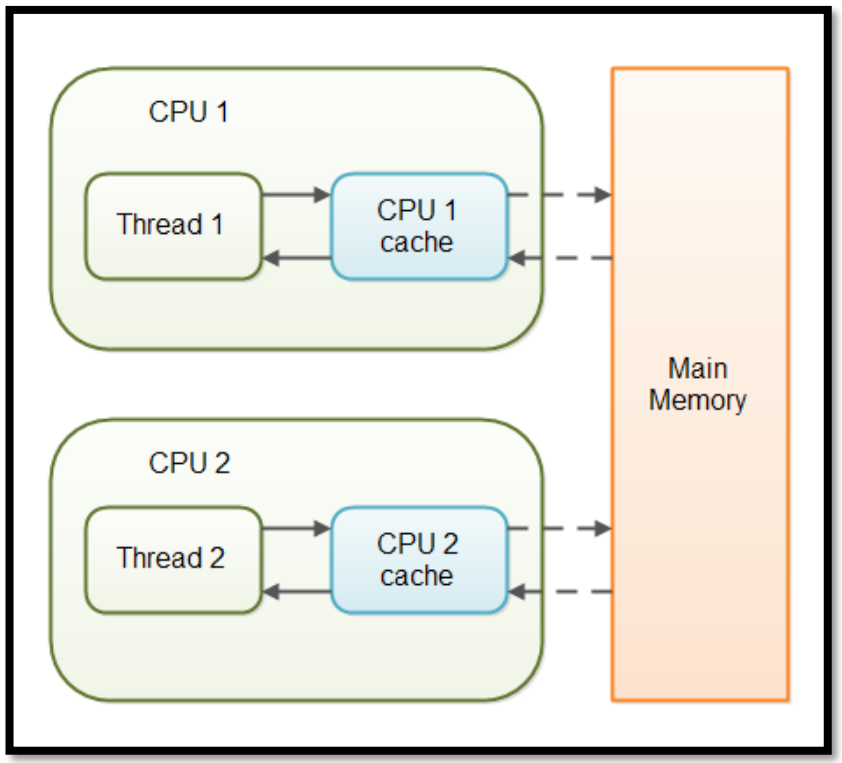
For **classes**, you can use either `final` or `abstract`:

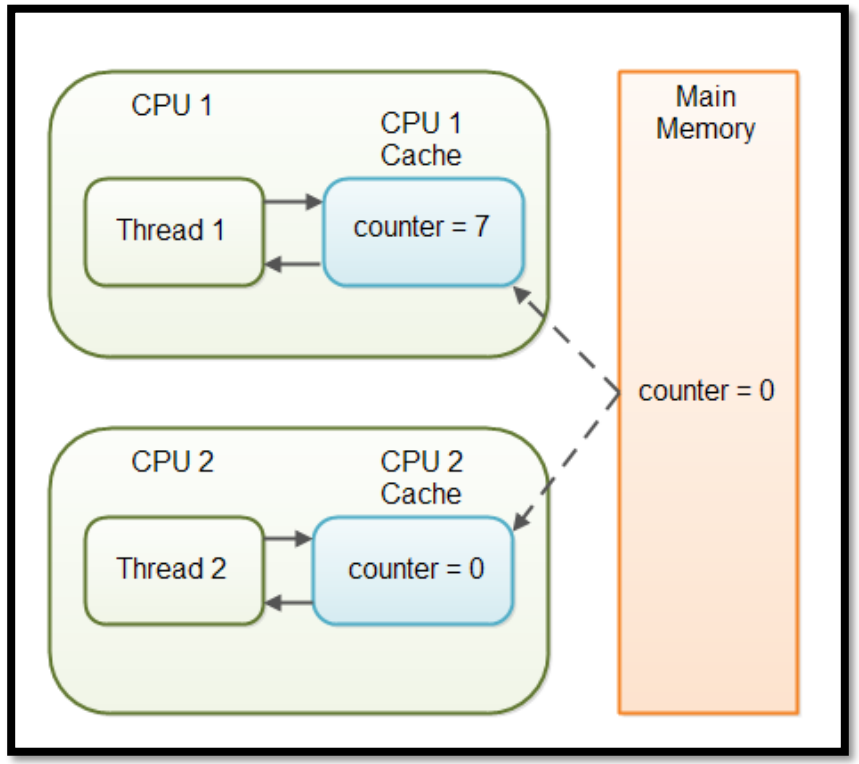
Modifier	Description	Try it
`final`	The class cannot be inherited by other classes (You will learn more about inheritance in the Inheritance chapter)	Try it »
`abstract`	The class cannot be used to create objects (To access an abstract class, it must be inherited from another class. You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters)	Try it »

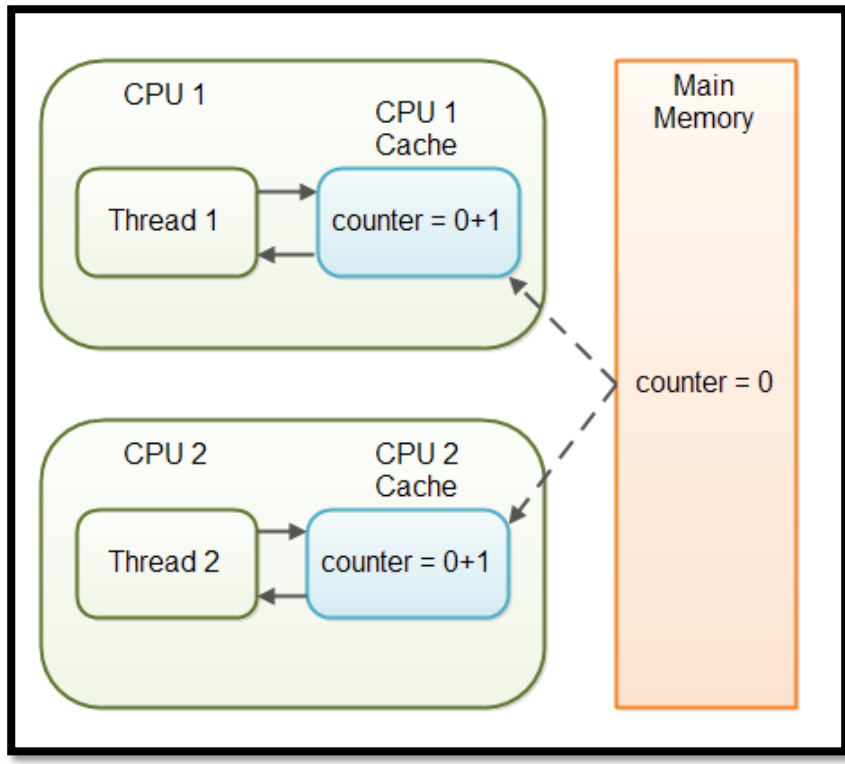
For **attributes and methods**, you can use the one of the following:

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). You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters
`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"

The Java volatile Visibility Guarantee







```
public class VolatileData
{
    private volatile int counter = 0;
    public int getCounter()
    {
        return counter;
    }
    public void increaseCounter()
    {
        ++counter;
    }
}
```

```
public class VolatileThread extends Thread
{
    private  VolatileData data;
    public VolatileThread(VolatileData data)
    {
        this.data = data;
    }
    @Override
    public void run()
    {
        int oldValue = data.getCounter();
        System.out.println("[Thread " + Thread.currentThread().getId() + "]:
                Old value = " + oldValue);
        data.increaseCounter();
        int newValue = data.getCounter();
        System.out.println("[Thread " + Thread.currentThread().getId() + "]:
                New value = " + newValue);
    }
}
```

Implication of this program…

```
public class VolatileMain
{
    private final static int noOfThreads = 3;
    public static void main(String[] args) throws InterruptedException
    {
        VolatileData volatileData = new VolatileData();
        Thread[] threads = new Thread[noOfThreads];
        for(int i = 0; i < noOfThreads; ++i)
                threads[i] = new VolatileThread(volatileData);
        for(int i = 0; i < noOfThreads; ++i)
                threads[i].start();
    }
}
```

```
[Thread 10]: Old value = 0
[Thread 12]: Old value = 0
[Thread 11]: Old value = 0
[Thread 12]: New value = 2
[Thread 10]: New value = 1
[Thread 11]: New value = 3
```

```
[Thread 10]: Old value = 0
[Thread 11]: Old value = 0
[Thread 12]: Old value = 0
[Thread 11]: New value = 2
[Thread 10]: New value = 1
[Thread 12]: New value = 3
```

```
[Thread 10]: Old value = 0
[Thread 12]: Old value = 0
[Thread 11]: Old value = 0
[Thread 12]: New value = 2
[Thread 10]: New value = 1
[Thread 11]: New value = 3
```