Encapsulation and Immutable Classes

Chapter 1

Encapsulation

Encapsulation = the ability to hide or protect an object's data.

```
// Hiding the attr to the outside world with the private keyword
private int myField = 0;

void setMyField(String val) { // Still accepting a String
    try {
        myField = Integer.parseInt(val);
    } catch(NumberFormatException e) {
        myField = 0;
    }
}
```

Access Modifier

You can apply these modifiers to classes, attributes and methods according to the following table:

	Class/ Interface	Class		Interface	
		Attrib	Method	Attrib	Method
public	X	X	X	X	X
private		X	X		
protected		X	X		
default	X	X	X	X	X

public = it can be accessed from any other class in our application
private = it can only be accessed inside the class that defines it. private is the most restrictive access modifier.
protected = it can be only accessed by the class that defines it, its subclasses and classes of the same package.
doesn't have a modifier = it has default access also known as package access, because it can only be accessed by classes within the same package.

```
package street21;
public class House {
    protected int number;
    private String reference;

    void printNumber() {
        System.out.println("Num: " + number);
    }
    public void printInformation() {
        System.out.println("Num: " + number);
        System.out.println("Ref: " + reference);
    }
}
class BlueHouse extends House {
    public String getColor() { return "BLUE"; }
}

...

package city;
import street21.*;
class GenericHouse extends House {
    public static void main(String args[]) {
        GenericHouse h = new GenericHouse();
        h.number = 100;
        h.reference = "";
        h.printNumber();
        h.printInformation();
        BlueHouse bh = new BlueHouse();
        bh.getColor();
    }
}
```

```
package street21;
public class House {
    protected int number;
    private String reference;

    void printNumber() {
        System.out.println("Num: " + number);
    }
    public void printInformation() {
        System.out.println("Num: " + number);
        System.out.println("Ref: " + reference);
    }
}
class BlueHouse extends House {
    public String getColor() { return "BLUE"; }
}

...

package city;
import street21.*;
class GenericHouse extends House {
    public static void main(String args[]) {
        GenericHouse h = new GenericHouse();
        h.number = 100;
        h.reference = ""; // Compile-time error
        h.printNumber(); // Compile-time error
        h.printInformation();
        BlueHouse bh = new BlueHouse(); // Compile-time error
        bh.getColor(); // Compile-time error
    }
}
```

Access Modifier

Here's a summary of the rules:

	Members same class	Subclass same package	Subclass different package	Another class same package	Another class different package
public	X	X	X	X	X
private	X				
protected	X	X	X	X	
default	X	X		X	

Singleton

Singleton = want to have only one instance for a particular class.

#1 Using a private static attribute and a method to get it

```
class Singleton {
    private static final Singleton instance = new Singleton();
    private Singleton() { }
    public static Singleton getInstance() {
        return instance;
    }
}
```

#2 Using a static inner class

```
class Singleton {
    private Singleton() { }
    private static class Holder {
        private static final Singleton instance =
                new Singleton();
    }
    public static Singleton getInstance() {
        return Holder.instance;
    }
}
```

The advantage of this is that the instance **won't be created until the inner class is referenced for the first time**.

Singleton

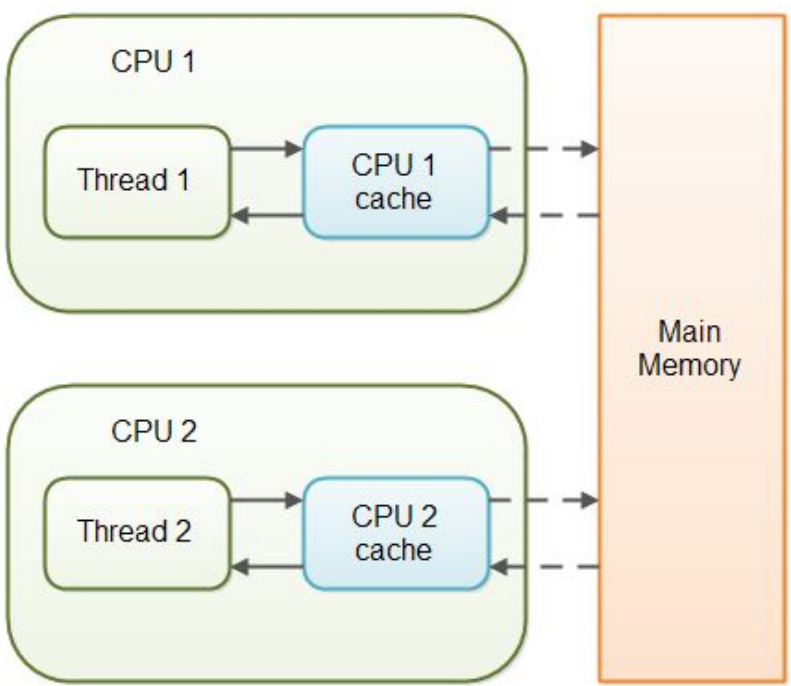
#3 Using a private static attribute and you want or have to defer the instantiation of the class until you first use it (also called lazy initialization)

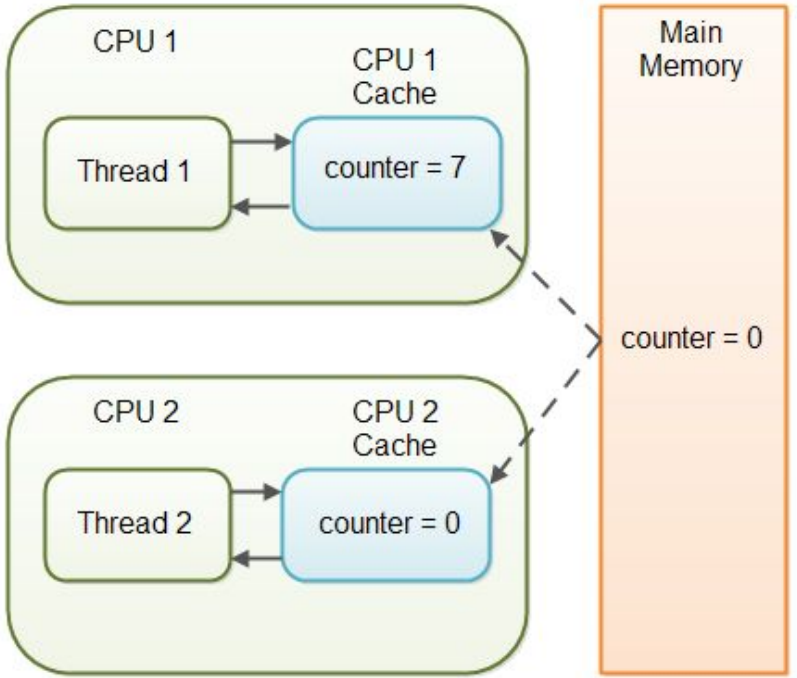
```
class Singleton {
    private static volatile Singleton instance;
    private Singleton() { }
    public static Singleton getInstance() {
        if(instance == null) {
            synchronized (Singleton.class) {
                if(instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

the **volatile** keyword, which guarantees that any read/write operation of a variable shared by many threads would be atomic and not cached.

Volatile

By declaring the counter variable volatile all writes to the counter variable will be written back to main memory immediately.
Also, all reads of the counter variable will be read directly from main memory.





Singleton

#4 Using enum

```
public enum SingletonEnum
{
    INSTANCE;

    public void doStuff()
    {
        System.out.println("Singleton using Enum");
    }
}
```

```
public class Singleton7 {
    public static void main(String[] args) {
        SingletonEnum.INSTANCE.doStuff();
    }
}

/*
1. You cannot create an instance of an enum by using the new operator (because the constructor is private).
2. An instance of an enum is created when the enum is first referenced.
3. An enum can't be reassigned.
4. enums are thread-safe by default
*/
```

Immutable Objects

Immutable object = you might not want to modify the values or state of an object when used by multiple classes.

There are some immutable classes in the Java JDK, for example:

- java.lang.String
- Wrappers classes (like Integer)

In summary, an immutable object:

- Sets all of its properties through a constructor
- Does not define setter methods
- Declares all its attributes private (and sometimes final)
- Has a class declared final to prevent inheritance
- Protects access to any mutable state. For example, if it has a List member, either the reference cannot be accessible outside the object or a copy must be returned (the same applies if the object's content must change)

The static Keyword

Think of something static as something belonging to the class and not to a particular instance of that class.

```
public class Example {
    private static int attr = 0;

    public void printAttr() {
        System.out.println(attr);
    }

    public static void main(String args[]) {
        Example e1 = new Example();
        e1.attr = 10;
        e1.printAttr();
        Example e2 = new Example();
        e2.printAttr();
    }
}
```

Output:

```
10
10
```

```
public class Example {
    private static int attr = 0;
    public static void printAttr() {
        System.out.println(attr);
    }
    public static void main(String args[]) {
        Example e1 = new Example();
        e1.attr = 10;
        e1.printAttr();
        // Referencing the method statically
        Example.printAttr();
    }
}
```

Output:

```
10
10
```

The static Keyword

A `static` (initializer) block looks like this:

```
public class Example {
    private static int number;

    static {
        number = 100;
    }
    ...
}
```

static blocks <u>cannot</u> contain a **return** statement

the keywords **super** and **this**, **<u>cannot</u>** be used in static method or static block initializer

Final Keyword

The final keyword can be applied to variables, methods, and classes.

When final is applied to variables, you cannot change the value of the variable after its initialization. These variables can be attributes (static and non-static) or parameters.

Final attributes can be initialized either
1. when declared
2. inside a constructor
3. inside an initializer block.

```java
public class Example {
    private final int number = 0;
    private final String name;
    private final int total;
    private final int id;

    {
        name = "Name";
    }

    public Example() {
        number = 1;
        total = 10;
    }

    public void exampleMethod(final int id) {
        id = 5;
        this.id = id;
    }
}
```

initialized either when declared, inside a constructor, or inside an initializer block.

```
public class Example {
    private final int number = 0;
    private final String name;
    private final int total;
    private final int id;

    {
        name = "Name";
    }

    public Example() {
        number = 1; // Compile-time error
        total = 10;
    }

    public void exampleMethod(final int id) {
        id = 5; // Compile-time error
        this.id = id; // Compile-time error
    }
}
```

Final Method

When `final` is applied to a method, this cannot be overridden.

```
public class Human {
    final public void talk() { }
    public void eat() { }
    public void sleep() { }
}
...
public class Woman extends Human {
    public void talk() { } // Compile-time error
    public void eat() { }
    public void sleep() { }
}
```

Final Class

When final is applied to a class, you cannot subclass it.
This is used when you don't want someone to change the behavior of a class by subclassing it.
Two examples in the JDK are java.lang.String and java.lang.Integer.

```
public final class Human {
    ...
}
...
public class Woman extends Human { // Compile-time error
    ...
}
```

In summary:

- A final variable can only be initialized once and cannot change its value after that.
- A final method cannot be overridden by subclasses.
- A final class cannot be subclassed.