- ***Singleton Design Pattern -- In Detail***

Scope: *Object Lifetime / Per Class loader*
The Singleton pattern is used for controlling the number of instances of a class (to one in general). This is maybe the first, the most used and the easiest design pattern out there.

Description:
The Singleton pattern is a way for multiple objects to share a common object without having to know whether it already exists.
The Singleton pattern ensures that only one instance of a class is created. All objects that use an instance of that class use the same instance.
There must be at least one instance of a class. Even if the methods of our class use no instance data or only static data, we may need an instance of the class for a variety of reasons.
The one instance of a class must be accessible to all clients of that class. These classes usually involve the central management of a resource. The resource may be external, as is the case with an object that manages the reuse of database connections. The resource may be internal, such as an object that keeps an error count and other statistics for a compiler.

Purpose:
Some of the more common reasons are that we need an instance to pass a method of another class or that we want to access the class indirectly through an interface.
There should be no more than one instance of a class. This may be the case because we want
to have only one source of some information.

Pros:

- Single point access to a single instance of a class, basically a sort of "globally" accessible instance of the class
- *Instance control* : Singleton prevents other objects from instantiating their own copies of the Singleton object, ensuring that all objects access the single instance.
- *Flexibility* : Because the class controls the instantiation process, the class has the flexibility to change the instantiation process.
- *Memory Consumption* : An object is cheap to create but takes up a lot of memory or continuously use other resources during its lifetime..

- ***Immutable Design Pattern -- In Detail***

Scope: Object Lifetime
An immutable class is simply a class whose instances cannot be modified. All of the information contained in each instance is provided when it is created and is fixed for the lifetime of the object.

Description:
When designing a new information holding class, we need to consider whether it

should be mutable (changeable) or immutable (not changeable), or maybe a little bit of both. There are some major pitfalls either way, but careful design can get rid of most of the problems.
There is a powerful and simple concept in programming that is really underused: Immutability
Basically, an object is immutable if its state doesn't change once the object has been created. Consequently, a class is immutable if its instances are immutable. The Immutable pattern increases the robustness of object that share references to the same object & reduces the overhead of concurrent access to an object.

Purpose:
Why are immutable objects so good then? There are many reasons for sure, here are the three main ones:

***Protection or solution* :**
You can send an immutable object to any class without worrying about it being altered by that class, and you never have to make a defensive copy. Same when you get one for local storage in your class (for instance a cache), you don't have to worry about whether the provider will hold on to a reference and change it later, invalidating your cache without you knowing about it.

***Performance*:**
Given point 1 you don't have to make defensive copies all the time. This means that you save the garbage collector some work which increases performance and decreases memory overhead, and we all want that don't we?

***Thread Safe*:**
After creation any number of threads can access them simultaneously, without any synchronization. This reinforces point 2 as well.

Pros:

- Since the state of immutable objects never changes, there is no need to write code to manage such changes.
- Instances of the immutable class are unique which means that tests for matches can be made using the == operator instead of the more expensive equals() method.
- Immutable objects are simple.
- Immutable objects are inherently thread-safe; they require no synchronization.
- Immutable objects can be shared freely.
- Not only can you share immutable objects, but you can share their internals.
- Immutable objects make great building blocks for other objects.
- The best use of the immutable objects is as the keys of a map.