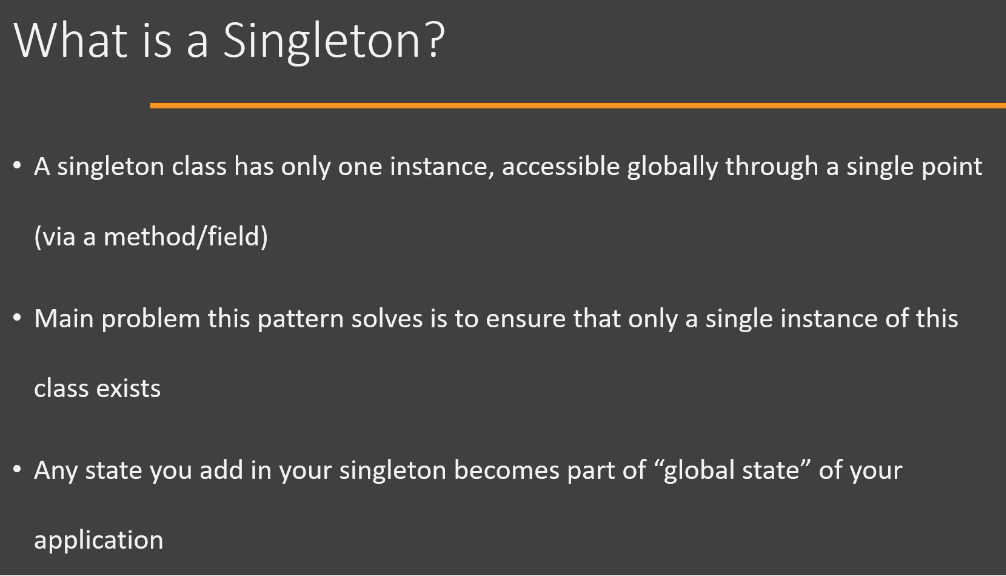
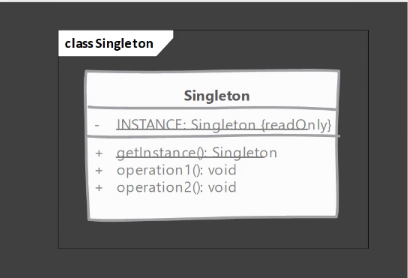
Singleton Design Pattern

1. Singleton – Introduction:

Now it's time to learn about Singleton design pattern. Now, this is a creational design pattern, and there is a good chance that you have already used this singleton design pattern in your code because this is one of the most used design pattern out there. So let's look at what exactly is a singleton. A singleton class or a class which implements Singleton Design pattern has only one instance in your program, and that instance is typically accessible globally through a single point. So you provide access to that one instance via either a method or a public field in your singleton class. Now, the main problem that this design pattern solves is that it ensures that there is only one instance of your class in your application. Now you should be aware that any state that you add in your singleton object becomes part of global state of your application because your singleton instance is shared globally. That means it is accessible anywhere in your application. All of its state is considered as a global state and I'm highlighting this point because generally having a large global state is an indication of a bad design or implementation.



Now let's look at the UML diagram. Now there is nothing to it. Basically, we have a simple class called as a Singleton. The only thing that you should remember is that we provide a static method typically called as getInstance(), and this method returns its only instance present in your application. Now, this class itself is typically responsible for creating that one instance.



A yellow note with text on it

Description automatically generated

1. Singleton – Implementations Steps:

Let's look at the steps that we need to follow in order to implement a singleton. So in order to ensure that there is only one instance, we need to control the instance creation. So we should make sure that our class constructors are not accessible outside of our class. We need to also make sure that subclassing is not allowed, because once you allow subclassing, then the creation of subclass is not in your control. Next, we need to make sure that we are keeping track of that single instance. And typically the class itself, the Singleton class itself is a good place to keep hold or keep track of that instance. Then we need to make sure that we are giving access to that instance through a public static method. Now we can also expose this instance as a final static field, but it doesn't work for all types of singleton implementations and we're going to see that in a little bit.

A screenshot of a computer program

Description automatically generated

So in order to implement Singleton, there are two types of implementations available. We can have an eager singleton. So eager Singleton is a singleton whose instance is created as soon as the class is loaded. So we do not wait for someone to actually ask for that instance. As soon as our class is loaded, we are going to go ahead and create our one instance. And the second option obviously is the opposite of this. We have a lazy singleton in a lazy singleton implementation. We do not create our instance unless and until somebody actually asks for it. And we're going to see how to implement both of these types of Singleton in our Java code.

A black and orange text

Description automatically generated

1. Singleton – Implementation – Eagar Singleton:

It's time to start implementing Singleton design pattern. So first of all, we're going to start by implementing Eager Singleton. Now, this is one of the simplest way that you can implement Singleton design pattern. As we discussed, the requirement for Singleton design pattern is that we want to make sure that there is only one instance of our class ever created. That means we want to prevent anybody from creating instance of our class. And to do that we are going to declare a constructor in our class(i.e., EagerRegistry), which is a private constructor. This will take care of inheritance as well. So since we have a private constructor, we cannot inherit from EagerRegistry class. Next, we want to create our singleton instance. So to do that, we are going to declare a private static final field, and it will be of type EagerRegistry, which is this class itself. I'm going to name this static variable as INSTANCE. And on the same line we are going to initialize our singleton instance. So this INSTANCE variable now holds the only instance that we are ever going to create of our Singleton class. Lastly, we need to provide a publicly accessible static method[i.e., getInstance()] that returns this INSTANCE to outside world. So anybody can call our getInstance() method and get access to our singleton instance. To check this, I'm going to go to another class called as Client. And we have a main() method here. So to do that, first of all, I'm going to make sure that I cannot create instance of this class(i.e., EagerRegistry) using the constructor. And this is because the constructor itself is not visible or since it is private, we cannot access it.

EagerRegistry registry = new EagerRegistry(); // not possible

So how should we get access of our singleton instance, that is using the static method that we have provided called as getInstance(). And this is the only way that we can get access of our singleton instance. Now we can call this getInstance() method multiple times and each time we are going to get the exact same instance.

1. Singleton – Implementation – Lazy Singleton:

Now it's time to implement the singleton design pattern but this time we are going to make or create a lazy singleton. That means our singleton instance will be created when somebody actually asks for our singleton instance. And to do that, we are going to start by, first of all, as you can guess, making our constructor private because we want to make sure that nobody can create instance of our class(i.e., LazyRegistryWithDCL) outside of this class. Next, we're going to declare a static variable in our class(i.e., LazyRegistryWithDCL), which will be of same type as this class. And this variable, the static variable is going to hold our singleton instance. Now we are going to provide our static method that will return the singleton instance to outside world. I'm going to name this method as getInstance(). Here we are going to return INSTANCE. But we have not created our instance yet. To do that, we need to make sure that there is no instance already present. So we are going to say if INSTANCE is null, that means the instance is not yet created, then let's create the instance. But we have to make sure that we are handling the synchronization issues. That means if two threads call this method simultaneously then if we do not use any kind of synchronization mechanism, then we will end up with creating two different objects of this class and we want to avoid that. So we are going to use the synchronize block and we are going to use the Class class which is associated with our LazyRegistryWithDCL. So now we have taken a lock on this entire class and as a double check locking, we have to make sure that we are going to check our INSTANCE for NULL once again because it may happen that two threads might call our getInstance() method at the same time and they will both see that INSTANCE is null. Now, as soon as we hit a synchronize block, one of the thread is going to get the lock which is associated with this block and start executing the code inside it. The second thread is going to wait on this line[i.e., synchronized (LazyRegistryWithDCL.class)]. So instead of checking once again, if we simply say that INSTANCE=new LazyRegistryWithDCL() inside the synchronization block, we are again going to run into same problems that both of these threads are going to create two separate objects.

// If you do like this. Separate objects will be created

public static EagerRegistry getInstance() {

if(INSTANCE == null){

synchronized (LazyRegistryWithDCL.class){

INSTANCE = new LazyRegistryWithDCL();

}

}

return INSTANCE;

}

To avoid that, we are going to do the null check for our INSTANCE variable once again in our synchronized block. So this is what is commonly referred to as a Double Check Locking. So we are checking two times, twice to make sure that INSTANCE is still null. When we get lock of this synchronized block and if it is still null then we are going to create our new instance. Now, one thing that you should note is that when multiple threads are referring to a variable then it is quite common that these threads will cache value of this variable in one of the CPU registers. To overcome that, Java has provided us with a keyword called volatile. So as soon as we declare our instance variable as volatile, it will indicate to these threads that they should not use the cached version of this variable's value. So every time they want to access the INSTANCE value, they will refer to the main memory and that way, we can ensure or we can guarantee that both of these threads that are coming to our synchronized block will get the latest value that is present in the memory. And this is the important point here that we need to make sure that we are using volatile and we are checking the INSTANCE for null twice. Next. we can come to our Client class. So this is how you can implement the lazy singleton using double check locking.

1. Singleton – Implementation – Initialization Holder:

Now, we are going to look at another way that we can implement Lazy Singleton. In previous lecture, we saw that we can use the double check locking and use volatile keyword in order to create our lazy singleton. But there is another way that we can implement this lazy singleton, and that is using what is called as Lazy Initialization Holder idiom. And we're going to see what exactly is that as we implement this. So to start off again, we're going to make our constructor private. So that takes care of inheritance as well as making sure that nobody can create instance of our class outside. Next, we are again going to need a variable that holds our singleton instance. And if you remember from previously(1st way of creating lazy singleton), we were declaring this INSTANCE variable in our class. But here instead of that, what we are going to do is we are going to declare a static inner class and let's call this class RegistryHolder. So we have an inner class in our LazyRegistryIODH class. It is a private static class. And inside this class(i.e., RegistryHolder), I'm going to declare my static variable INSTANCE, which will be our variable that holds on to our singleton instance. Now we are going to initialize this variable on the same line. And then we are going to have our public static method as usual, that returns our singleton instance to outside world. Method name is getInstance() and we are going to return singleton instance[i.e., RegistryHolder.INSTANCE]. So remember, we are referring to this inner class (i.e., RegistryHolder) now, and we are going to return the INSTANCE that is declared inside(i.e., RegistryHolder). Let's take a quick look at what is happening here(inside LazyRegistryIODH class). So if you remember when we were implementing the Eager Singleton, inside EagerRegistry class, since our instance of Singleton class was initialized as an initialization of static variable, as soon as somebody loaded or caused our class loader to load this class(i.e., EagerRegistry), then the class loader itself was going to initialize this static variable(i.e., INSTANCE) with our new instance of Singleton class. And that's why it is called as an eager singleton, because before anybody called our getInstance() method, our instance was getting created. Now, here(i.e., LazyRegistryIODH class), if you look at it, we have not declared the static INSTANCE variable in our LazyRegistryIODH class. Instead, we have an inner class, a private static inner class, and that has this static variable. So applying the same logic, as soon as the class loader loads our RegistryHolder class, it will initialize this INSTANCE variable and create our singleton instance. But the only way anybody can access this class(i.e., RegistryHolder) or make the class loader load this is if somebody calls our getInstance() method because inside this method we are referring or making the first reference to our RegistryHolder. So that way, we are indeed using lazy initialization. So this class(i.e., RegistryHolder) won't be initialized unless and until somebody calls the getInstance() method. To prove that, what we are going to do is, in our constructor of this singleton class(i.e., LazyRegistryIODH), we are going to print some message. So I'm going to print 'In LazyRegistryIODH singleton'. So anytime our constructor gets called, this System.out will be printed. Now let's jump to our Client code and inside main() method, just keep 'LazyRegistryIODH singleton;'. We are just defining a reference variable of type LazyRegistryIODH. I'm not going to do anything else inside the main() method. I'm simply going to make a reference so that the class loader now has to load our LazyRegistryIODH class. So let's run this code as Java application and you will see that nothing has been printed in our console. Add one more line[i.e., System.out.println("Done");] inside main() method at the bottom. So I'm going to run once again. You will see that we get 'Done' printed but our constructor of Singleton class[i.e., LazyRegistryIODH] didn't print anything. That means the constructor was not called. Now watch what happens as soon as I call the getInstance() method of LazyRegistryIODH class. So as soon as I call the LazyRegistryIODH.getInstance() method, you will see that, now, our constructor was called because we see this line(i.e., 'In LazyRegistryIODH singleton') in the output. And you can keep calling this getInstance() method multiple times. Our constructor of LazyRegistryIODH class will be called only once. Put the following lines in main() method and run. Constructor will be called only once when the inner class is loaded.

LazyRegistryIODH singleton;

singleton = LazyRegistryIODH.getInstance();

singleton = LazyRegistryIODH.getInstance();

singleton = LazyRegistryIODH.getInstance();

singleton = LazyRegistryIODH.getInstance();

So this is called as Lazy Initialization Holder Idiom. And here we make use of an inner static class, which is private. And that way we make sure that even though we have initialized the static INSTANCE variable with our singleton instance, it won't be created unless and until somebody calls the getInstance() method. One thing just I wanted to quickly show that you cannot make reference to this inner class(i.e., RegistryHolder) from outside[like, LazyRegistryIODH.RegistryHolder holder;]. You don't get to access this class(i.e., RegistryHolder) from outside of LazyRegistryIODH class. So if I try to say, [LazyRegistryIODH.RegistryHolder holder;], you will see that this code won't compile because it will say that the inner class, since it is private, is not visible. So there is no way for anyone to refer to this inner class. And that way we get benefit of lazy Singleton. One quick change that you will see from our singleton with double check locking is that we don't need to provide any synchronization and perform any double check locking or use volatile in order to get the benefit of Lazy Singleton. All right, so this is a singleton implementation using lazy initialization holder idiom.

1. Singleton – Implementation – Enum:

Now we're going to look at one last way that you can implement singleton, and that is using 'enum'. So this was presented by Joshua Bloch in the Google I/O 2008. And it is also mentioned in the 'Effective Java' book that you can read. So here instead of creating our singleton as a regular class, we are going to create an enum(i.e., RegistryEnum) and we are going to declare our singleton INSTANCE as an enum constant, and then we can keep on adding any methods that you would typically add in your class. For example, something like getConfiguration(), so on and so forth. So having your singleton as an enum takes care of a couple of problems. First of all, any class cannot extend/inherit from any enum(i.e., because all Enums implicitly extend java.lang.Enum class. Since Java does not support multiple inheritance, an enum cannot extend anything else). So that means you don't have to worry about inheritance. Secondly, you cannot create objects of enum in your class. The only object of this enum(i.e., RegistryEnum) is going to be this INSTANCE constant that we have declared. So it takes care of having a single instance created. One thing that using Enum provides is that it also handles **serialization** and **deserialization** issues because using regular classes that we have seen so far, if you deserialize a previously serialized singleton instance, you will get another different object. So that way you will end up with two objects of your same singleton class. But I have never seen a need for serializing a singleton instance, so I haven't covered that in much detail. And I believe you won't ever need to serialize your singleton instance. But this is something that you should know that this is out there, that you can use an Enum as a singleton. The benefit that it provides is that it handles the Deserialization issue, so it won't create a new instance if you are using an enum. But I don't see anybody using serialization with Singleton.

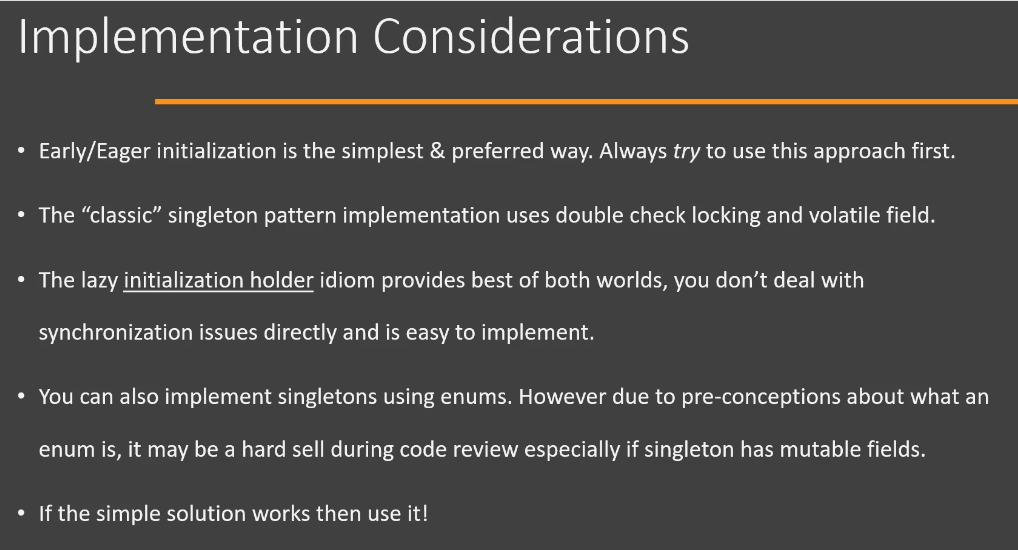
The best preferred way out of all that we have seen, I would highly recommend to stick with the Lazy Initialization Holder idiom because this pattern provides you best of both worlds. You don't have to deal explicitly with synchronization issues and using volatile, you still get to have your instance created when actually when somebody actually needs it. And if this is also too much for you, you can go for Eager singleton where we are creating our singleton instance as soon as class is loaded and the best way is not to use Singleton at all because it is considered an anti-pattern. So we're going to discuss what that anti-pattern is in the pitfalls of this design pattern.

Note on Enum in Java:

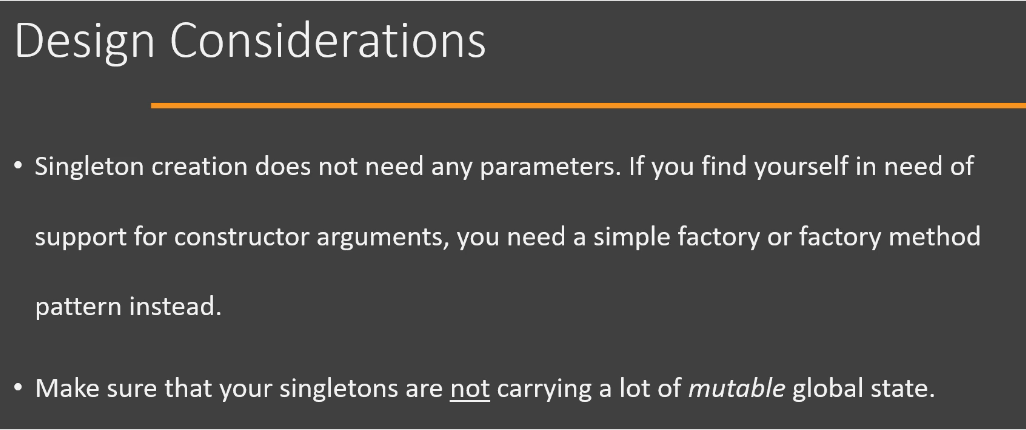
An enum can, just like a class, have attributes/fields and methods. The only difference is that enum constants are public, static and final(unchangeable - cannot be overridden). We can’t create the instance of the Enum keyword because it contains only private constructors, and it cannot extend other classes (but it can implement interfaces).

1. Singleton – Implementation & Design Considerations:

It's time to look at some of the implementation and design considerations while working with singleton. Now, first of all, using an eager singleton is the most simplest way, and that should be your preferred way and you should start out your singleton implementation by this approach. And then if you see any issues with your startup time and you need to make sure that your startup time should be less, then only you should go for other approaches. Now the classic singleton pattern that we saw, which is using the double check locking and volatile field, is the one that you will see a lot of times implemented in the real world. The lazy initialization holder idiom that we saw as a second way of implementing the lazy singleton. It provides best of both worlds to you. That means you don't have to explicitly handle synchronization issues since you are depending on the class loader to handle those issues on your behalf. And you also get a benefit that your Singleton instance is now created when somebody actually calls your getInstance() method, the lazy initialization holder is the preferred way that I would suggest you to use if you want to have a lazy singleton. Now you can also implement singletons using Enums. You can declare your singleton as an enum and add methods inside it, but due to the preconceptions about what an Enum stands for, it could be a hard sell during code review, especially if your singleton has any mutable state because enums are thought to be as constants. So if you have a singleton declared as an enum, then it better have a immutable state inside it. Lastly, I would like to point out that if simple solution works, then go with it. You don't have to handle synchronization and use volatile if eager Singleton works for you.

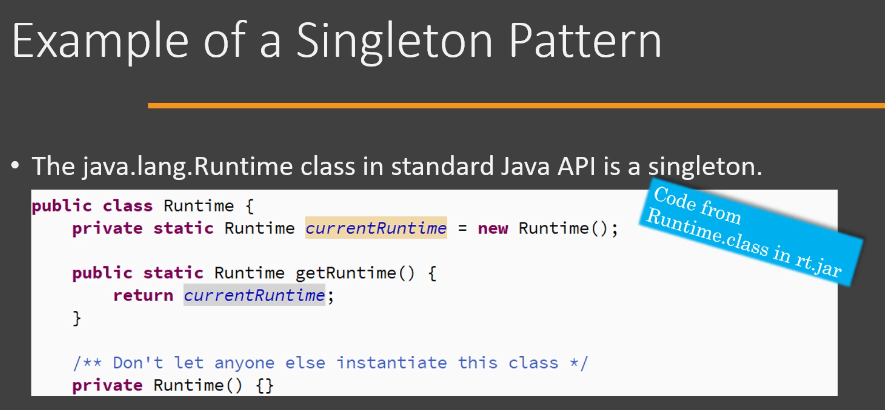


Now let's look at some of the design considerations when working with Singleton. So Singleton creation itself doesn't need any parameters or any arguments from outside world. So your getInstance() method should not expect any arguments to be passed. In case you are running into a situation where you think that you might need arguments to create your first object, then you should instead use either a simple factory or factory method design pattern. Now, one thing to note here is that you have to make sure that your singletons do not carry around a lot of mutable global state because Singleton, as it is, is considered an anti-pattern nowadays. That means it is a pattern that you should avoid using in your code because it creates a lot of problems. And we're going to see some of those problems when we are discussing pitfalls of singleton pattern. So the point here that I want to highlight is that if you are using Singleton, then at least make sure that the state of your singleton is not mutable. Because it(singleton instance) becomes part of your application's global state. And having mutable global state is just a recipe for many, many bugs during the development and maintenance. So these are some of the points that you should keep in mind when using Singleton design pattern.



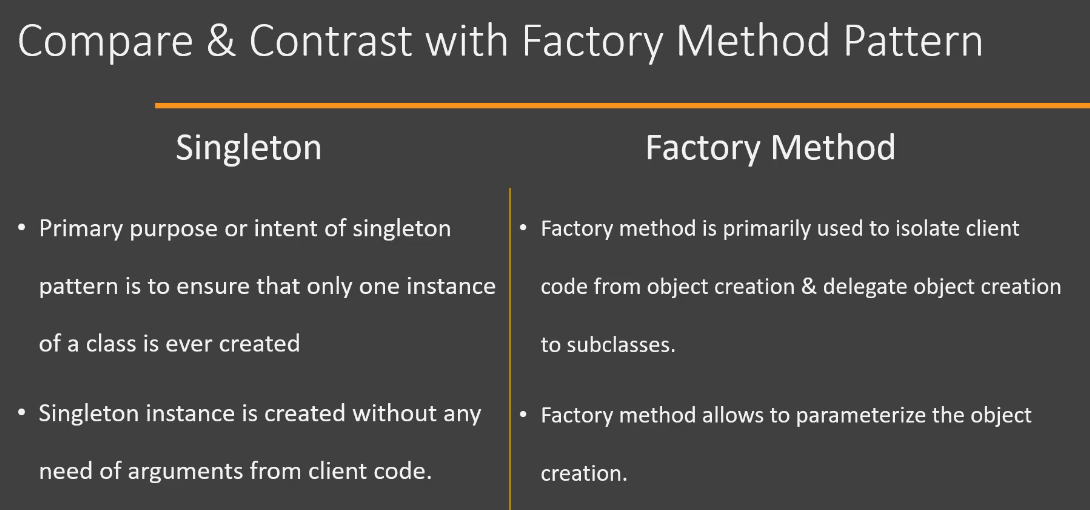
1. Example of a Singleton design pattern in Java:

Now let's look at the example of Singleton design pattern in Java class library and great example of this design pattern, where it is actually good to have or use the Singleton design pattern, is the Runtime class which is present in the Java class library. So it is part of the standard Java API and below is the code snippet from that class. This code is taken from Runtime class in Java file and you will see that in this class we find the typical characteristic of singleton design pattern. First characteristic being that we have a private constructor. Second, you will see that we have a private static variable where our singleton instance will be stored and a public static method which is used by the external code to get hold/access of that singleton instance. So you will see that the Runtime class is actually implemented as an Eager singleton. So java.lang.Runtime class is an example of singleton design pattern in real world.



1. Singleton – Comparison with Factory Method:

Now let's compare our Singleton design pattern with Factory Method design pattern. Now the intent behind Singleton design pattern is to ensure that only one instance of our class is ever created. The Factory Method, on the other hand, has main concerns related to separation of client code from the object creation and delegation of object creation to subclasses. So the purpose behind these two design patterns is quite different. Even though both of these belong to Creational design Pattern category. Next, a singleton instance is created without need of any arguments from client code. So as we saw, we have a single method called as getInstance() and that is called anytime we want access to our singleton instance and that method doesn't need any arguments. On the other hand, we can parameterize the object creation in a factory method. All right, so these are the differences between singleton and factory method design patterns.



1. Singleton – Pitfalls:

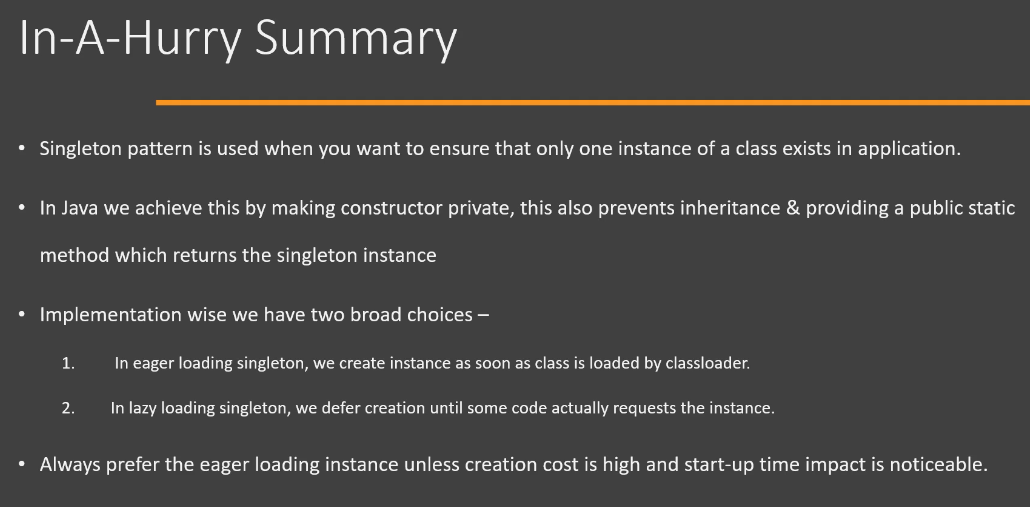
Now let's look at the pitfalls of Singleton design pattern. And there are quite a few of those. First of all, Singleton design pattern can deceive you about true dependencies of your code. Since the Singleton instance is available globally, using the getInstance() method, different parts of your code can start depending on that singleton instance, and this dependency is not quite obvious. Second pitfall is that singletons are difficult to unit test because we have a static method and it is difficult to mock the instance that is returned by that static method. Next, one of the common ways to implement singleton design pattern is that we use a static variable to hold on to our singleton instance, but the static variables are created per class loader and not per JVM. So if you are running your application in a web container like Tomcat, your singleton may not remain a singleton in that single JVM. So if you have a singleton class which is used or deployed in two separate web applications which are running in a single Tomcat instance, then you have two singleton instances in that single JVM. Now, this is typically not a problem, but if your singleton is tied to an external resource which is unique, then you should keep in mind that static variable means static or one copy per class loader, not per JVM[**Note:** If you load the same class using different classloaders, each class will have its own static variables. Each of these fields will be unique per-classloader. If you have the class loaded by multiple classloaders in a JVM**]**. Next, a singleton class which has a lot of mutable global state, is actually a good indication that you are misusing Singleton design pattern. And this is one of the main reasons that singleton design pattern is avoided nowadays because having a mutable global state is considered a bad design practice.

A screenshot of a computer

Description automatically generated

1. Singleton – Summary:

Now we will look at summary of Singleton design pattern. So we use singleton pattern whenever we want to ensure that there is only one instance of our class in entire application. To implement Singleton in Java, we start by making our constructor private. That way, nobody outside of our class can create instances of our class and nobody can extend our class. Then we provide a public static method in our Singleton class in order to get our singleton instance outside. Now, when we are implementing Singleton, there are two broad choices available to us. We can implement our singleton as an eager singleton. An eager singleton is a singleton where we create our one instance as soon as our class is loaded by the class loader. On the other hand, we have lazy singleton, and here we create our instance when somebody actually asks for that instance. Now you should always try to use eager singleton because it avoids many complexities related to synchronization. However, you should note that creating or implementing eager singleton adds to your application startup time, because as soon as class loader loads your class, you will be creating your instance. And if the cost of creating the instance is high, then it will add up to your application startup time. So if you are worried about your startup time, then you should look into lazy Singleton.



**Note:**

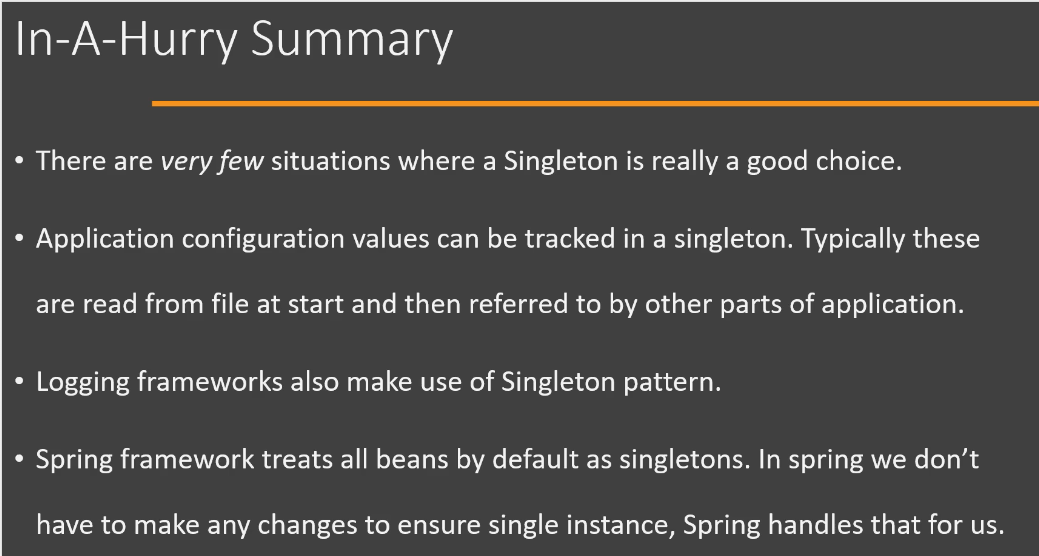
So a volatile variable is not cached by threads. So if multiple threads are accessing this variable's value, then they will refer to the main memory in order to check what the current value of this variable is.

A screenshot of a computer program

Description automatically generated

**Note:**

We can implement Singleton as an enum, so we will simply declare a public enum. You will declare your INSTANCE as the enum constant and you will provide all the methods that you would typically implement inside of the class. Now, having your singleton as an enum takes care of all the problems that you need to take care of if you are implementing Singleton using a regular class.



Now, there are actually very few situations where a singleton design pattern is a good choice. Singleton is one of those design pattern which is overused and often misused. One of the situations where Singleton is a good choice is for storing the application configuration values. So in this situation, we will have a file on our file system which will store the configuration values, provided by customers, used by our application. Now we can have a singleton class which reads these values at the application startup and provides access to these values from multiple parts of our code. Now, one important point is that these values should be immutable. That means you should not allow changing your configuration values from within your code because if you do then you will run into the same problem of having mutable global state. Logging frameworks like log4j also make use of singleton design pattern. Then there is Spring framework, where we have an IOC(Inversion of Control) container and a great framework for enterprise development. Now inside Spring application, we define our Java classes as spring beans. Now spring will instantiate these beans and provide them with dependencies. Now, by default, all the beans that are created by spring will be singleton. One benefit here is that since the instance creation is handled by spring, we do not have to modify our code in order to make it singleton. Spring also supports other types of instantiation models, but Singleton is the default way in which spring creates beans.

So this is all about the Singleton design pattern in Java.

Why is singleton considered an anti-pattern? 🡪 Singletons are easy to overuse & abuse. Any mutable state of a singleton is now a global state. A large global state makes it harder to maintain application as it can be modified from any part of application.

What is double check locking that is used in lazy singleton implementation? 🡪 We check twice for the singleton reference to be null, once before synchronization block and then inside the synchronization block. We have to make sure the reference is declared as volatile. We check reference initially to see if the thread has to initialize singleton. If it is null then thread tries to acquire the lock & may get blocked if another thread is inside block. That is why once we get lock, we must check if another thread has created the instance or not.

Is it mandatory to *always* override clone() method in singleton to avoid duplicating objects? 🡪 No! unless our singleton implements Cloneable(which we won't do for 'singleton'), calling clone will just throw an exception and not create any copy. Some websites suggest you to override clone(). But remember clone() is protected in java.lang.Object class. So unless the client code is declared in java.lang package they can't call clone() method. Next, clone() method only clones an object if the class is implementing Cloneable. We're writing a singleton here, why would we ever implement Cloneable? We don't need to always override clone() in singleton class.