# Overview:

* The **Singleton** pattern is one of the simplest patterns in Java:
  + Comes under creational pattern as this pattern provides one of the best ways to create an object.
* Ensure a class only has one instance, and provide a global point of access to it.
* We are taking a class and letting it manage a single instance of itself.
  + Also preventing any other class from creating a new instance on its own.
  + To get an instance, you have got to go through the class itself.
* We are also providing a global access point to the instance:
  + Whenever you need an instance, just query the class and it will hand you back the single instance.
  + A global variable makes an object accessible, but it does not keep you from instantiating multiple objects.

# Examples:

* Although there can be many printers in a system, there should be only one printer spooler.
* There should be only one file system and one window manager.
* An accounting system will be dedicated to serving one company.
* Logging, driver’s objects, caching, and thread pool would also use a **Singleton**.
* The **Singleton** design pattern is used in core java classes also, for example java.lang.Runtime, java.awt.Destop.

# Advantages & disadvantages of the singleton:

* Controlled access to sole instance:
  + Because the **Singleton** class encapsulates its sole instance, it can have strict control over how and when clients access it.
* Reduced name space:
  + An improvement over global variables.
  + Avoids polluting the name space with global variables that store sole instances.
* Permits a variable number of instances:
  + Makes it easy to change your mind and allow more than one instance of the **Singleton** class.
* **Singletons** hinder unit testing :
  + Might cause issues for writing testable code if the object and the methods associated with it are so tightly coupled that it becomes impossible to test without writing fully-functional class dedicated to the **Singleton**.
* **Singleton**s create hidden dependencies:
  + Because it is readily available throughout the code base, it can be overused.
  + Since its reference is not completely transparent while passing to different methods, it becomes difficult to track.

# Overview of implementation:

* When this method is called, it checks to see whether the object has already been instantiated :
  + If it has, the method just returns a reference to the object.
  + If not, the method instantiates it and returns a reference to the new instance.

# Singleton vs Dependency Injection (Overview):

* There is some discussion in the software community on when you should utilize a **Singleton** and when you should utilize dependency injection.
* We know dependency injection is a technique whereby object supplies the dependencies of another object:
  + Enables you to replace dependencies without changing the class that uses them.
* Dependency injection can also be used to avoid statics (one of the most common reasons to use it).
* We know that **Singletons** ensure only one instance of an object.
* Using DI, you can use constructor or setter injection to pass around a single object.
  + Have the injector create a single object and then inject it via the constructor or setter of any dependent objects.
  + Implements the **Singleton** with less dependencies.

# Singleton dependencies:

* We know that there are some disadvantages of utilizing the **Singleton** pattern.
* **Singletons** create hidden dependencies:
  + Because it is readily available throughout the code base, it can be overused.
  + Since its reference is not completely transparent while passing to different methods, it becomes difficult to track.
* **Singletons** hinder unit testing:
  + Might cause issues for writing testable code if the object and the methods associated with it are so tightly coupled that it becomes impossible to test without writing a fully-functional class dedicated to the **Singleton**.
  + Use of static makes hard to test.
* So, the question becomes, can dependency injection solve the disadvantage of the **Singleton** pattern’s introduction of new dependencies.
  + Can DI reduce/eliminate these dependencies?
* The answer is “it depends on the situation”, in some cases, dependency injection is preferred over **Singletons**.

# When to use DI over the Singleton?

* When you want your software to be under unit test:
  + It is much easier to write unit tests for your code when using DI (less coupling).
* When you want to avoid using statics:
  + Statics make code harder to test.
  + **Singletons** use statics.
* When you have a non-stable dependency.
* A non-stable dependency is a dependency which refers to or affects the global state, such as an external service, file, system, or a database.
  + It is good practice to inject that dependency to the dependent class.
  + Helps the class explicitly specify everything it needs in order to perform properly.
* Using DI over **Singletons** is mostly, but not entirely, about testing.
* Sometimes you do not want a **Singleton**, just more flexible code.
* There still are dependencies which are better represented using a **Singleton**.
* Ambient dependencies are dependencies which span across multiple classes and often multiple layers.
  + The **Singleton** pattern is better suited to handle this.
    - You do not want to pass the injector object to all of these multiple classes.
* A good example of when to use a **Singleton** over DI is a logger service:
  + If you tend to log a lot of activities throughout your code base, it just not practical to pass the logger instance to ever class that needs it as a dependency.
* It is important to keep a balance between the dependencies represented as **Singletons** and the ones injected using the DI principles.
* If a dependency is ambient, meaning that I is used by many classes and/or multiple layers, use a **Singleton**.
* Otherwise, inject it to the dependent classes using the DI principle.

# Implementation overview:

* To implement the **Singleton** pattern, there are different approaches but all of the have the following concepts:
  + Private constructor to restrict instantiation of the class from other classes.
  + Private static variable of the same class that is the only instance of the class.
  + Public static method that returns the instance of the class.
    - Is the global access point for outer world to get the instance of the **Singleton** class?

# Approaches (Five):

* We will discuss five main approaches when implementing the **Singleton** pattern:
  + **Lazy evaluation approach:**
    - It’s not multi-thread safe.
    - Use this approach if you are not worried about multiple threads.
    - This is not a recommended approach.
  + **Synchronized approach:**
    - Thread safe.
    - Use when performance is not critical to your application, but, it is multi-threaded.
    - Straightforward and effective.
  + **Double-checked locking principle approach:**
    - Thread safe.
    - Increases performance from the synchronized approach.
  + **Eager evaluation approach:**
    - If your application always creates and uses an instance of the **Singleton**.
    - Does not uses a lot of resources.
    - Thread safe.
    - The instance is created even though client application might not be using it.
  + **Bill Pugh approach:**
    - Thread safe.
    - High performance.
    - Ensures that the instance is only created if a client needs it.
    - Create the **Singleton** class using an inner static helper class.
    - Regarded as the standard method to implement **Singletons** in Java.

# Approach details:

# Problems with Lazy Evaluation Approach:

* The implementation on the previous slides are not thread safe.
* Suppose two calls to **getInstance ()** are made at virtually the same time.
* The first thread checks to see whether the instance exists. It does not, it goes into the part of the code that will create the first instance.
* However, before it has done that, suppose a second thread also looks to see whether the instance member is null:
  + Because the first thread has not created anything yet, the instance is still equal to null, so the second thread also goes into the code that will create an object.
* Both threads new perform a new on the **Singleton** object, thereby creating two objects.
* If the **Singleton** is absolutely stateless, then thread safety may not be a problem.
* If the **Singleton** has state, and if you except that when one object changes the state, all other objects should see the change, then this could become a serious problem.
* Inconsistent state between threads using the different **Singleton** objects.
* If the object creates a connection, there will actually be two connections (one for each object).
* If a counter is used, there will be two counters.
* It may be very difficult to find these problems.
  + Dual creation is very intermittent (it usually won’t happen).
  + It may not be obvious why the counts are off, because only one client object will contain one of the **Singleton** objects while all the other client objects will refer to the other **Singleton**.

# Problems with Synchronized approach:

* One big problem is that the synchronization may end up being a severe bottleneck.
  + All the threads will have to wait for the check on whether the object already exists.
  + Reduces the performance because of cost associated with the synchronized method.
* The only time synchronization is relevant is the first time through this method.
  + Once we have set the **uniqueInstance** variable to an instance of **Singleton**, we have no further need to synchronize this method.
  + After the first time through, synchronization is totally unneeded overhead.
* For most Java applications, we need to ensure that the **Singleton** works in the presence of multiple threads and does not have performance issues.
  + Use the double checked locking principle.

# Double Checked Locking principle:

* This approach will use a synchronized block inside the if condition with an additional check to ensure that only one instance of the **Singleton** class is created.
  + Intent is to optimize away unnecessary locking, increase performance.
  + The synchronization check happens at most one time, so it will not be a bottleneck.
* Use “double-checked locking” to reduce the use of synchronization in **getInstance ()**.
* With double-checked locking, we first check to see if an instance is created, and if not, then we synchronize.
  + We only synchronize the first time through, just what we want.

# Eager Initialization approach:

* In eager initialization, the instance of **Singleton Class** is created at the time of class loading.
  + The easiest method to create a **Singleton** class.
  + It has a drawback that the instance is created even though client application might not be using it.
* Using this approach, we rely on the JVM to create the unique instance of the **Singleton** when the class is loaded.
  + The JVM guarantees that the instance will be created before any thread accesses the static **uniqueInstance** variable (thread safe).

# Static Block Initialization:

* Since static blocks will be called only once, we can use static blocks to develop singleton class.
* Static block Initialization implementation is similar to eager initialization, except that instance of class is created in the static block that provides option for exception handling.
* Both eager initialization and static block initialization creates the instance even before it’s being used and that is not the best practice to use.

# Singleton (Eager Initialization) problems:

* If your **Singleton** class is not using a lot of resources, then the eager initialization is the approach to use.
* In the most of the scenarios, **Singleton** classes are created for resources such as File System, Database connections etc.
  + We should avoid the instantiation until/unless client calls the **getInstance** method.
* So, Bill Pugh came up with a different approach to create the **Singleton** class using an inner static helper class.
* This is the most widely used approach for the **Singleton** class.
  + Does not require synchronization, is thread safe and only creates the instance when the client needs it.
  + It is easy to understand and implement.
  + Regarded as the standard method to implement **Singletons** in Java.

# Bill Pugh approach:

* In the most of the scenarios, **Singleton** classes are created for resources such as File System, Database connections etc.
  + We should avoid the instantiation until/unless client calls the **getInstance** method.
* So, Bill Pugh came up with a different approach to create the **Singleton** class using an inner static helper class.
* This is the most widely used approach for the **Singleton** class:
  + Does not require synchronization, is thread safe and only creates the instance when the client needs it.
  + It is easy to understand and implement.
  + Regarded as the standard method to implement **Singletons** in Java.

# Enum Singleton:

* To overcome this situation with Reflection, Joshua Block suggests the use of Enum to implement Singleton design patterns as Java ensures that any **Enum** value is instantiated only once in a Java program. Since Java **Enum** values are globally accessible, so is the singleton. The drawback is that the Enum type is somewhat inflexible. For example, it does not allow lazy initialization.
* **Enum Singletons are easy to write:** If you have been writing Singletons before Java 5, this is by far the greatest benefit that you realize that you can have more than one instance even with double checked locking. While this problem is solved by improving the Java memory model and guaranteeing volatile variables from Java 5 onwards, it is still difficult for many beginners to write. Compared to double-check locking with synchronization, Enum singletons are very easy. If you don’t think that the following code for traditional singleton with double-checked locking and Enum Singletons re compared:
  + Singleton using Enum in Java: By default creation of the Enum instance is thread-safe, but any other Enum method is the programmer’s responsibility.
* **Enum Singletons handle Serialization by themselves:** Another problem with conventional Singletons is that they are no longer Singleton once you implement a serializable interface because the method **readObject ()** always returns a new instance just like the Java constructor. By using the **readResovle ()** method and discarding newly created instances, you can avoid that by substituting Singleton.
  + If your Singleton Class maintains state, this can become even more complex, as you need to make them transient, but **JVM** guarantees serialization with **Enum Singleton**.
* **Creation of Enum instance is thread-safe:** by default, the Enum instance is thread-safe, and you don’t need to worry about double-checked locking.
* **Coding Constraints:** In regular classes, there are things that can be achieved but prohibited in Enum classes. Accessing a static field in the constructor, for example. Since he’s working at a special level, the programmer needs to be more careful.
* **Serializability:** For singletons, it is very common to be stateful. In general, those singletons should not be serializable. There is no real example where transporting a stateful singleton from one **VM** to another **VM** makes sense. A **Singleton** means “Unique within a **VM**” not “Unique in the universe”. If serialization really makes sense for a stateful singleton, the **Singleton** should specify explicitly and accurately what it means in another **VM** to deserialize a **Singleton** where there may already be a **Singleton** of the same type.

# Summary:

* Use the lazy evaluation approach for creating a **Singleton** if you are not worried about multiple threads:
  + Not recommended.
* For most Java applications, you will need to ensure that the **Singleton** works in the presence of multiple threads.
  + Add the synchronized keyword to the **getInstance ()** method if performance is not critical to your application.
  + Synchronizing getInstance () is straightforward and effective.
    - Just keep in mind that synchronizing a method can decrease performance by a factor of 100, so if a high-traffic part of your code begins using getInstance (), you may have to reconsider.
  + Use the double-checked locking principle to increase performance in the getInstance () method.
* If your application always creates and uses an instance of the **Singleton** and is not using a lot of resources.
  + Move to an eagerly created instance approach rather than a lazily created one.
  + Will be thread safe.
  + Has a drawback that the instance is created even though client application might not be using it.
* So, the recommended approach is the Bill Pugh method:
  + Create the **Singleton** class using an inner static helper class.