It is Creation-al Design Pattern

SingleTon Type	How to Break
1. Eager Initialization and breaking it with reflection API	

1. private static final BasicConnectionPool *instance* = new BasicConnectionPool();
2. // Public static method to get the instance
public static BasicConnectionPool getInstance() {
return *instance*;
}
3.
	We can break this basic eager initialization using reglection api
	Eg:
	• private static void breakUsingReflection() {
	try {
	BasicConnectionPool instance1 = BasicConnectionPool.*getInstance*();
	
	// Using reflection to break singleton
	Constructor<BasicConnectionPool> constructor =
	
	BasicConnectionPool.class.getDeclaredConstructor();
	constructor.setAccessible(true);
	BasicConnectionPool instance2 = constructor.newInstance();
	
	System.*out*.println("instance1 hash: " + instance1.hashCode());
	System.*out*.println("instance2 hash: " + instance2.hashCode());
	System.*out*.println("Are instances same? " + (instance1 == instance2));
	}catch (Exception e){
	e.printStackTrace();
	}
	}
	•

1.1→ Breaking of eager initialization singleton using Reflection is resolved using instance !=null check inside constructor	

Solution to eager initialization breaking using reflection →	Add below inside constructor
	• // Private constructor to prevent instantiation
	private BasicConnectionPoolWithReflectionBreakResolved() {
	
	// Protection against reflection
	if (*INSTANCE* != null) {
	throw new IllegalStateException("Instance already created");
	}
	System.*out*.println("Creating SecureConnectionPool instance");
	}

2. breaking eager Initialized Singleton class with Serialization and deserialization	

- How breaking work with serialization and deserialization
- We serialize instance to a file using *ObjectOutputStream*
- *This writes the object's state to a byte stream*
- *We then deserialize it back into a new object using ObjectInputStream*
- *This creates a new instance by reading the object's state from the byte stream*
- *The JVM doesn't use the constructor; instead, it creates a new instance using reflection*
-
	Breaking with seraliation and deserialization :
	• private static void breakWithSerialization() {
	try {
	// Get the singleton instance
	BasicConnectionPoolWithSerializationBreakResolved instance1 =
	BasicConnectionPoolWithSerializationBreakResolved.*getInstance*();
	
	// Serialize the instance to a file
	try (ObjectOutput out = new ObjectOutputStream(
	new FileOutputStream("singleton.ser"))) {
	out.writeObject(instance1);
	}
	
	// Deserialize it back into a new object
	try (ObjectInput in = new ObjectInputStream(
	new FileInputStream("singleton.ser"))) {
	BasicConnectionPoolWithSerializationBreakResolved instance2 =
	(BasicConnectionPoolWithSerializationBreakResolved) in.readObject();
	
	// Print hash codes to check if they're the same instance
	System.*out*.println("instance1 hash: " + instance1.hashCode());
	System.*out*.println("instance2 hash: " + instance2.hashCode());
	System.*out*.println("Are instances same? " + (instance1 == instance2));
	}
	} catch (Exception e) {
	e.printStackTrace();
	}

2.2 Breaking of eager initialization singleton using serialization and deserialization is resolved by adding readResolve method in SingleTon class	

Resolution to serialization and deserialization break:	Just add below method to the singleton class
	// This method is called during deserialization
	@Serial
	protected Object readResolve() {
	return *getInstance*(); // Always return the existing instance
	}

3. → Lazy initialization and breaking it with Multi threaded env Lazy initialization steps:	

Problem: This breaks in multi-threaded environments because multiple threads can pass the null check simultaneously.	
Lazy initialization → here we create private instance	// Private static instance variable private static BasicConnectionPoolWithLazyInitialization *INSTANCE*;

Then we make private Constructor so that we prevent object creation outside of the class	// Private constructor to prevent instantiation
	private BasicConnectionPoolWithLazyInitialization() {
	
	    // Protection against reflection
	    if (*INSTANCE* != null) {
	      throw new IllegalStateException("Instance already created");
	    }
	  System.*out*.println("Creating SecureConnectionPool instance");
	}

Then in public static getInstance method:
- we check if instance == null
- then we create a new instance
- and we return the instance
	// Public static method to get the instance
	public static BasicConnectionPoolWithLazyInitialization getInstance() {
	  if (*INSTANCE* == null) {
	    INSTANCE = new BasicConnectionPoolWithLazyInitialization();
	  }
	  return *INSTANCE*;
	}

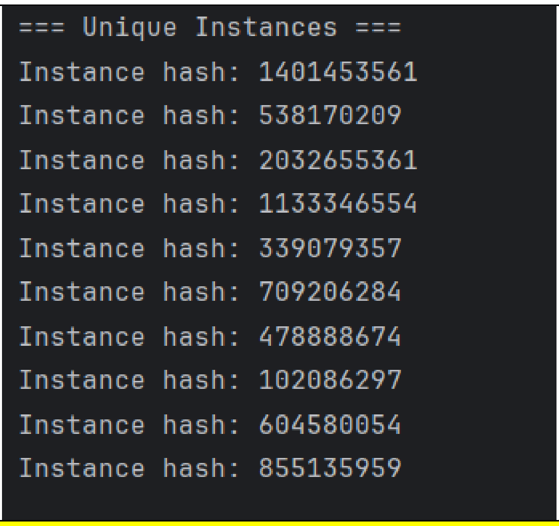
Also we add readResolve method to avoid duplication object creation in serialization and deserialization	// This method is called during deserialization
	@Serial
	protected Object readResolve() {
	  return *getInstance*(); // Always return the existing instance
	}

3.1 breaking Lazy initialized SingleTon class inside multi threaded Env Breaking steps:	

Creates multiple threads (10 by default)
Each thread tries to get the singleton instance
Stores all obtained instances in a synchronized set
After all threads complete, checks if only one unique instance was created
If multiple unique hash codes appear, the singleton is not thread-safe
The test uses CountDownLatch to ensure all threads start at approximately
	private static void breakLazyInitializedSingletonUsingMultiThreadedEnv() {
	
	    int numberOfThreads = 10;
	
	Set<BasicConnectionPoolWithLazyInitializationWithBillPughMethodUsingHolderStaticInnerClass> instances =
	        Collections.*synchronizedSet*(new HashSet<>());
	
	    CountDownLatch latch = new CountDownLatch(numberOfThreads);
	
	    for (int i = 0; i < numberOfThreads; i++) {
	      new Thread(() -> {
	        try {
	          // All threads will try to get the instance at the same

* the same time, increasing the chance of catching thread-safety issues	time
	
	BasicConnectionPoolWithLazyInitializationWithBillPughMethodUsingHolderStaticInnerClass instance =
	
	BasicConnectionPoolWithLazyInitializationWithBillPughMethodUsingHolderStaticInnerClass.*getInstance*();
	instances.add(instance);
	System.*out*.println("Thread " + Thread.*currentThread*().getId() +
	" got instance with hash: " +
	System.*identityHashCode*(instance));
	} catch (Exception e) {
	e.printStackTrace();
	} finally {
	latch.countDown();
	}
	}).start();
	}
	
	try {
	latch.await(); // Wait for all threads to complete
	} catch (InterruptedException e) {
	Thread.*currentThread*().interrupt();
	}
	
	// Print all unique instances
	System.*out*.println("\n=== Unique Instances ===");
	instances.forEach(instance ->
	System.*out*.println("Instance hash: " + System.*identityHashCode*(instance))
	);
	
	System.*out*.println("\nNumber of unique instances created: " + instances.size());
	
	if (instances.size() > 1) {
	System.*out*.println("\n Singleton pattern broken! Multiple instances were created.");
	} else {
	System.*out*.println("\n Singleton pattern maintained. Only one instance was created.");
	}
	
	}

After running this, we can see multiple instance got created by different threads



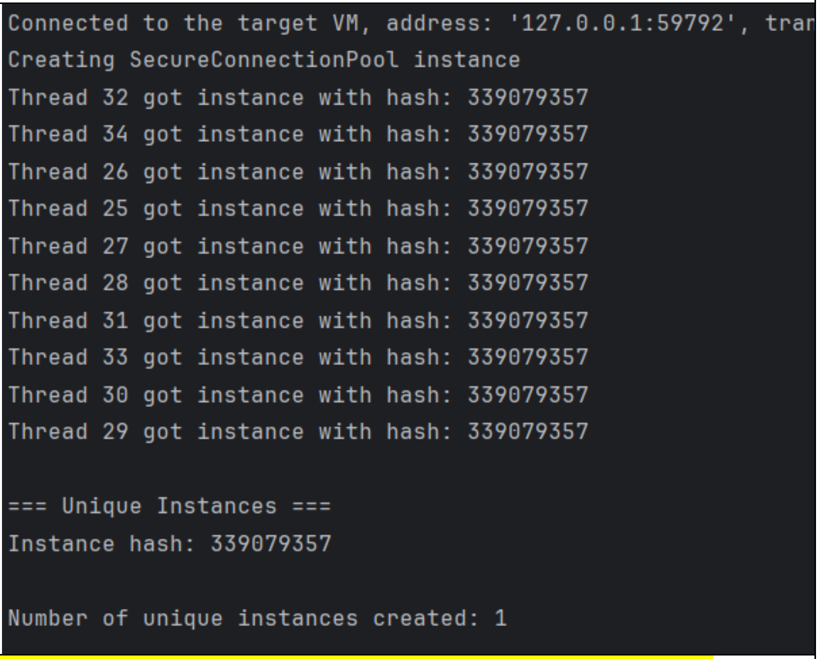
```
=== Unique Instances ===
Instance hash: 1401453561
Instance hash: 538170209
Instance hash: 2032655361
Instance hash: 1133346554
Instance hash: 339079357
Instance hash: 709206284
Instance hash: 478888674
Instance hash: 102086297
Instance hash: 604580054
Instance hash: 855135959
```

3.2 preventing to break lazy initialized singleton using single checked locking (by making whole getInstance method synchronized)

Step:

```
// Added synchronized keyword
  public static synchronized BasicConnectionPoolWithLazyInitialization getInstance() {
      if (INSTANCE == null) {
          INSTANCE = new BasicConnectionPoolWithLazyInitialization();
      }
      return INSTANCE;
  }
```

Since we have made whole getInstance method as synchronized, we are able to create only single instance →



```
Connected to the target VM, address: '127.0.0.1:59792', tran
Creating SecureConnectionPool instance
Thread 32 got instance with hash: 339079357
Thread 34 got instance with hash: 339079357
Thread 26 got instance with hash: 339079357
Thread 25 got instance with hash: 339079357
Thread 27 got instance with hash: 339079357
Thread 28 got instance with hash: 339079357
Thread 31 got instance with hash: 339079357
Thread 33 got instance with hash: 339079357
Thread 30 got instance with hash: 339079357
Thread 29 got instance with hash: 339079357

=== Unique Instances ===
Instance hash: 339079357

Number of unique instances created: 1
```

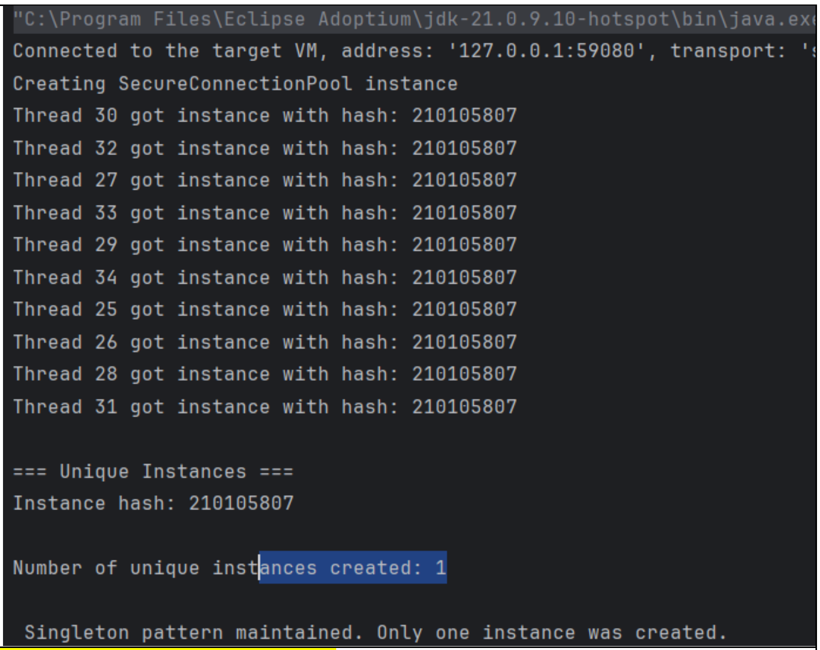
3.3 making whole getInstance method as synchronized is not a good idea and good practice because of below reasons

1. **Performance Overhead**:
    - Every call to getInstance() acquires and releases a lock, even after the instance is created
    - This can create a significant bottleneck in high-concurrency scenarios

	- Most calls only need to read the instance, not create it 2. **Unnecessary Synchronization**: - After the first call, the instance exists and won't change - Synchronization is only needed during the first call when creating the instance - Subsequent calls don't need synchronization but still pay the performance cost
3.4 instead of making whole getInstance method as synchronized, we can add double checked locking Double checked locking steps **Overall Benefits** - **Performance**: Only synchronizes the first time - **Thread Safety**: Guarantees single instance - **Efficiency**: Best of both worlds - safe and fast - **Lazy Initialization**: Instance created only when needed	
Add volatile keyword to the instance volatile **Keyword** - **What it does**: Ensures changes are visible to all threads immediately - **Why needed**: Prevents threads from seeing a partially constructed object - **Benefit**: Thread safety without full method synchronization	// Private static volatile instance variable private static volatile BasicConnectionPoolWithLazyInitializationDoubleCheckEdLockingUsingVolatile *INSTANCE*;

Add this double-checked locking in getInstance method
 	
First Null Check (Outside Synchronized)
- **What it does**: Quick check without locking
	public static BasicConnectionPoolWithLazyInitializationDoubleCheckEdLockingUsingVolatile getInstance() {
	  if (*INSTANCE* == null) { // First check (no locking)
	    synchronized (BasicConnectionPoolWithLazyInitializationDoubleCheckEdLockingUsingVolatile.class) {
	      if (*INSTANCE* == null) { // Second check (with locking)
	        INSTANCE = new BasicConnectionPoolWithLazyInitializationDoubleCheckEdLockingUsingVolatile();

• **Why needed**: Improves performance
• **Benefit**: Skips synchronization for all calls after instance creation
 	
3. Synchronized Block
• **What it does**: Locks the class during instance creation
• **Why needed**: Prevents multiple threads from creating instances
• **Benefit**: Thread safety during the only time it's needed (first creation)
 	
4. Second Null Check (Inside Synchronized)
• **What it does**: Verifies again if instance is null
• **Why needed**: Ensures only one thread creates the instance
• **Benefit**: Prevents multiple instances if multiple threads pass first check
	}
	}
	}
	return *INSTANCE*;
	}

If you see, only one instance is created if we use double checked locking	"C:\Program Files\Eclipse Adoptium\jdk-21.0.9.10-hotspot\bin\java.ex Connected to the target VM, address: '127.0.0.1:59080', transport: ' Creating SecureConnectionPool instance Thread 30 got instance with hash: 210105807 Thread 32 got instance with hash: 210105807 Thread 27 got instance with hash: 210105807 Thread 33 got instance with hash: 210105807 Thread 29 got instance with hash: 210105807 Thread 34 got instance with hash: 210105807 Thread 25 got instance with hash: 210105807 Thread 26 got instance with hash: 210105807 Thread 28 got instance with hash: 210105807 Thread 31 got instance with hash: 210105807 === Unique Instances === Instance hash: 210105807 Number of unique instances created: 1 Singleton pattern maintained. Only one instance was created.

4 → breaking lazy initialized singleton using Clone method

Steps for breaking

Why This Happens:

- **The clone() method creates a new instance of the object**
- **This bypasses the singleton pattern's single-instance guarantee**
- **The hash codes will be different, proving two instances exist**

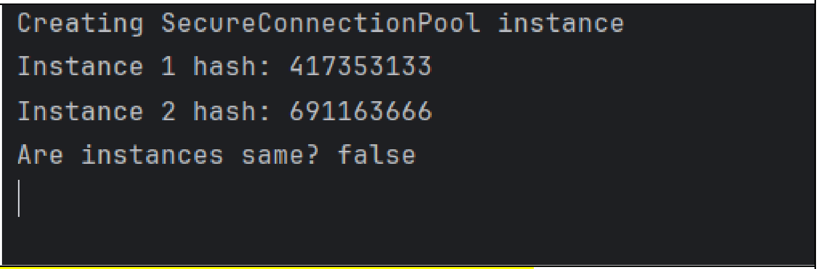
Step1: your singleton class implements Cloneable interface	public class BasicConnectionPoolWithLazyInitializationCheckingWithCloneable implements Serializable , Cloneable{

Step 2: your singleton class overrides object class clone method	@Override
	protected Object clone() throws CloneNotSupportedException {
	    return super.clone(); // This can create a new instance
	}

Step 3: pro proving the breaking, first you get the singleton instance → then calls clone method and created another instance → and if you check hashcode of these 2 instances, you will see different hashcode is getting created	private static void breakSingletonWithClone() {
	    try {
	        // Get the singleton instance
	BasicConnectionPoolWithLazyInitializationCheckingWithCloneable instance1 =
	BasicConnectionPoolWithLazyInitializationCheckingWithCloneable.*getInstance*();
	        // Create a clone of the singleton
	BasicConnectionPoolWithLazyInitializationCheckingWithCloneable instance2 =

```
(BasicConnectionPoolWithLazyInitializationCheckingWithCloneable
) instance1.clone();

        // Print hash codes
        System.out.println("Instance 1 hash: " +
System.identityHashCode(instance1));
        System.out.println("Instance 2 hash: " +
System.identityHashCode(instance2));
        System.out.println("Are instances same? " + (instance1 ==
instance2));

    } catch (CloneNotSupportedException e) {
        System.out.println("Clone not supported: " + e.getMessage());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Step 4: verification confirms that singleton class is breaked with 2 different instances are got created since we can clearly see 2 different hash code



```
Creating SecureConnectionPool instance
Instance 1 hash: 417353133
Instance 2 hash: 691163666
Are instances same? false
```

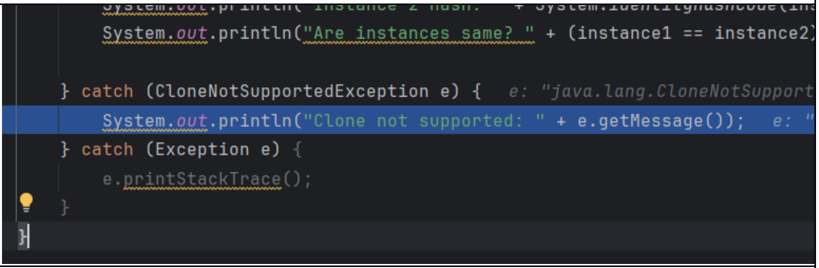
4.1 → preventing cloning and creating separate instances, we need to throw CloneNotSupportedException from overridden clone method inside our singleton class

Steps:

Throw CloneNotSupported exception from overridden clone method inside your singleton class

```
@Override
protected Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException("Singleton instance
cannot be cloned");
}
```

You can see , when you tried to clone, you got the exception, and cloning is prevented



```
        System.out.println("Instance 2 hash: " + System.identityHashCode(ins
        System.out.println("Are instances same? " + (instance1 == instance2)

    } catch (CloneNotSupportedException e) {  e: "java.lang.CloneNotSupport
        System.out.println("Clone not supported: " + e.getMessage());  e: "
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

5.Initialization-on-demand holder idiom (Bill Pugh's solution):

enum singleton pattern is **unbreakable** in Java. Here's why:
No Reflection: The JVM prevents creating enum instances via reflection
No Cloning: java.lang.Enum has a final clone() method that prevents cloning
No Deserialization Issues: The JVM handles enum deserialization specially to return the same instance

No Class Loading Tricks: The JVM guarantees enum instances are created only once, even across multiple class loaders

No Constructor Access: Enum constructors are private and cannot be called directly

This is why Joshua Bloch (author of "Effective Java") recommends the enum approach as the best way to implement a singleton in Java. It's the only approach that is guaranteed to be a true singleton in all scenarios.

1. Create Enum with Single Instance	public enum BestSingleton {
	INSTANCE; // Single enum instance
	}

2. Add Private Constructor
Note: by default enum Constructor is private, no need to add private keyword, if we add, it will be redundant
	private BestSingleton() {
	// Initialization code here
	}

3. Add Connection Pool	private final ConnectionPool connectionPool = new ConnectionPool();

4. Create ConnectionPool Inner Class	private static class ConnectionPool {
	private final List<Connection> connections = new ArrayList<>();
	private static final int MAX_POOL_SIZE = 10;
	}

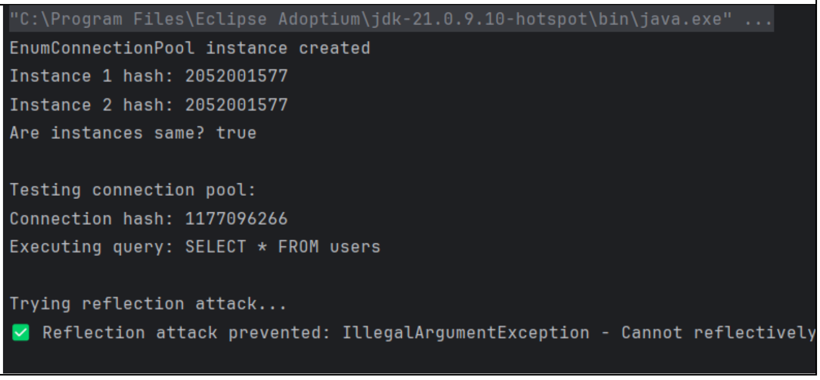
5. Initialize Connections	private ConnectionPool() {
	for (int i = 0; i < MAX_POOL_SIZE; i++) {
	connections.add(new Connection());
	}
	}

6. Add Thread-Safe getConnection()	public synchronized Connection getConnection() {
	for (Connection conn : connections) {
	if (!conn.isInUse()) {
	conn.setInUse(true);
	return conn;
	}
	}
	throw new RuntimeException("No available connections");
	}

7. Add releaseConnection() Method	public synchronized void releaseConnection(Connection conn) {
	conn.setInUse(false);
	}

8. Add Public Methods to Enum	public Connection getConnection() {
	return connectionPool.getConnection();
	}
	
	public void releaseConnection(Connection conn) {
	connectionPool.releaseConnection(conn);
	}

- **9. overriding clone method is not needed because**	Overriddig clone and throwing CloneNotSupportedException Is not needed

java.lang.Enum already has a final clone() method that throws CloneNotSupportedException	
10. Add Serialization Support (Handled Automatically by Enum)	- No need for readResolve() or writeReplace() Enums handle serialization automatically
If you see → using Bill Pugh, enum singleton, even if multiple instances are created, then their hash is same	 "C:\Program Files\Eclipse Adoptium\jdk-21.0.9.10-hotspot\bin\java.exe" ... EnumConnectionPool instance created Instance 1 hash: 2052001577 Instance 2 hash: 2052001577 Are instances same? true Testing connection pool: Connection hash: 1177096266 Executing query: SELECT * FROM users Trying reflection attack... ✅ Reflection attack prevented: IllegalArgumentException - Cannot reflectively

1)

- singleton - which means you can create only 1 instance of the class for the entre JVM

- if I create 2 different instances of the class, then we can say, both classes will different values

Real world example :

a. Connection pooling object in database (just imagine what will happen to your database if multiple

objects are created for connection class , everyone will connect to db, and connection to db will break at some point)

b. cache manager (it doesn't make sense to store multiple objects in cache, create a single object, store data and access it from all places writhing the application)

c. Logger

d. Making a report lets say for junit test cases, you want to construct a report

e. Thread pool (Efficiently manage a pool of threads for concurrent tasks.)

if you create multiple instances of these , then memory waste/leak problem can occur

2)it provides single point of access to the class- generally using getInstance method

3)

Types of SingleTonclass:

1)Eager initialisation

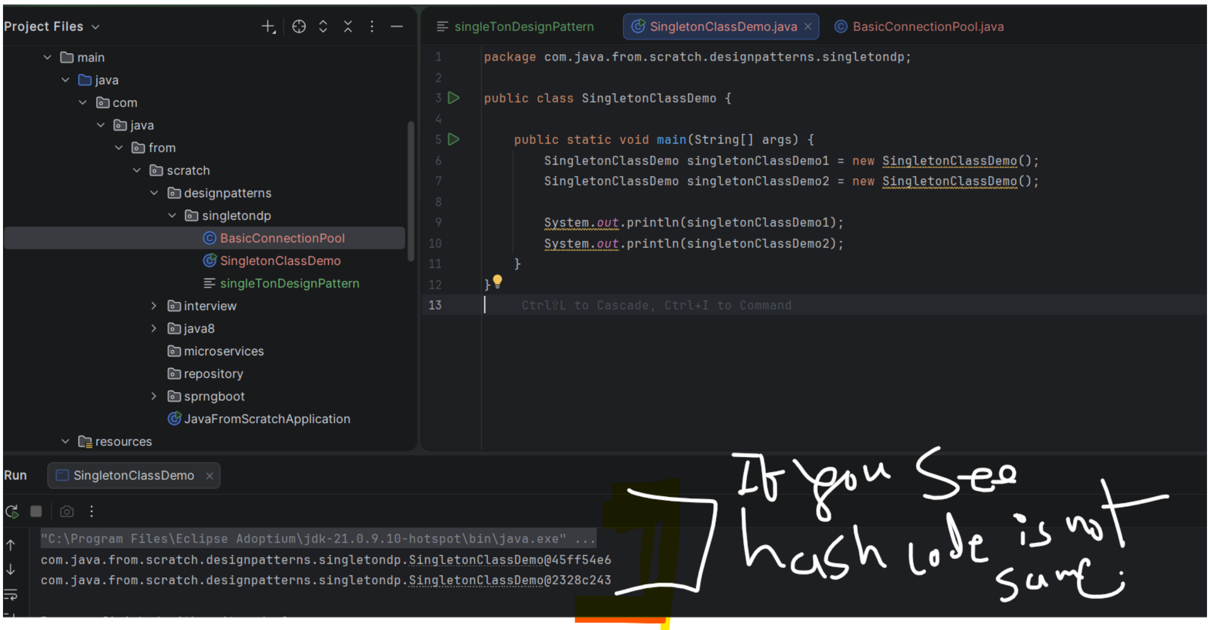
2)Lazy initiation

4)

Eager initialisation)

```
/* SingletonClassDemo singletonClassDemo1 = new SingletonClassDemo();
 SingletonClassDemo singletonClassDemo2 = new SingletonClassDemo();

 System.out.println(singletonClassDemo1);
 System.out.println(singletonClassDemo2);*/
```

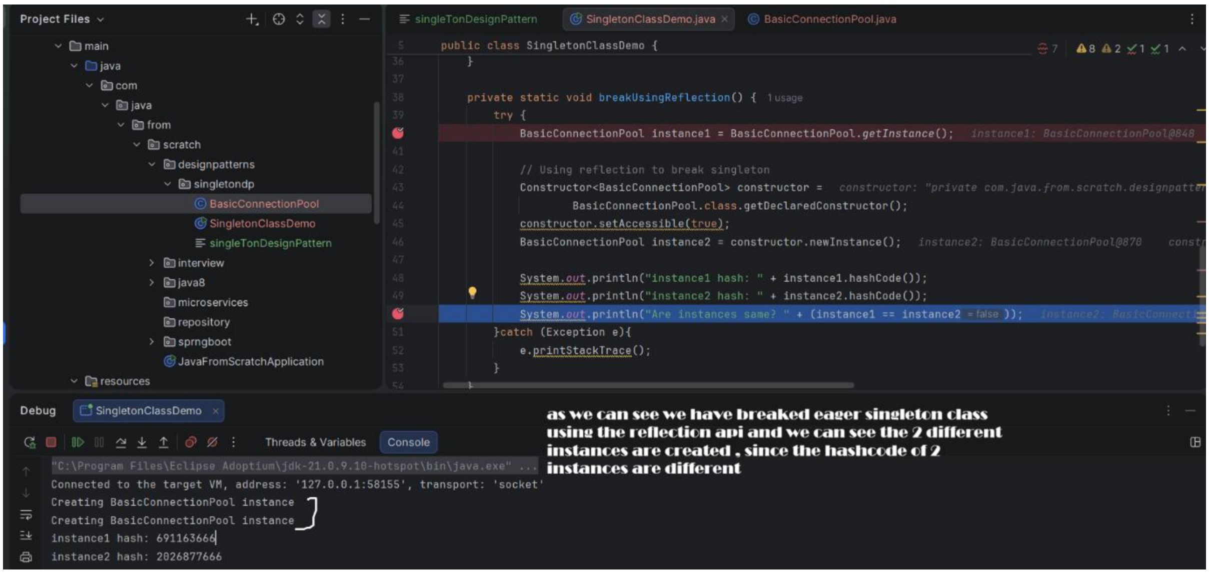
These 2 instances should retun the same hashcode



```
package com.java.from.scratch.designpatterns.singletondp;

public class SingletonClassDemo {

    public static void main(String[] args) {
        SingletonClassDemo singletonClassDemo1 = new SingletonClassDemo();
        SingletonClassDemo singletonClassDemo2 = new SingletonClassDemo();

        System.out.println(singletonClassDemo1);
        System.out.println(singletonClassDemo2);
    }
}
    Ctrl⇧L to Cascade, Ctrl+I to Command
```

```
"C:\Program Files\Eclipse Adoptium\jdk-21.0.9.10-hotspot\bin\java.exe" ...
com.java.from.scratch.designpatterns.singletondp.SingletonClassDemo@45ff54e6
com.java.from.scratch.designpatterns.singletondp.SingletonClassDemo@2328c243
```

If you See hash code is not same;



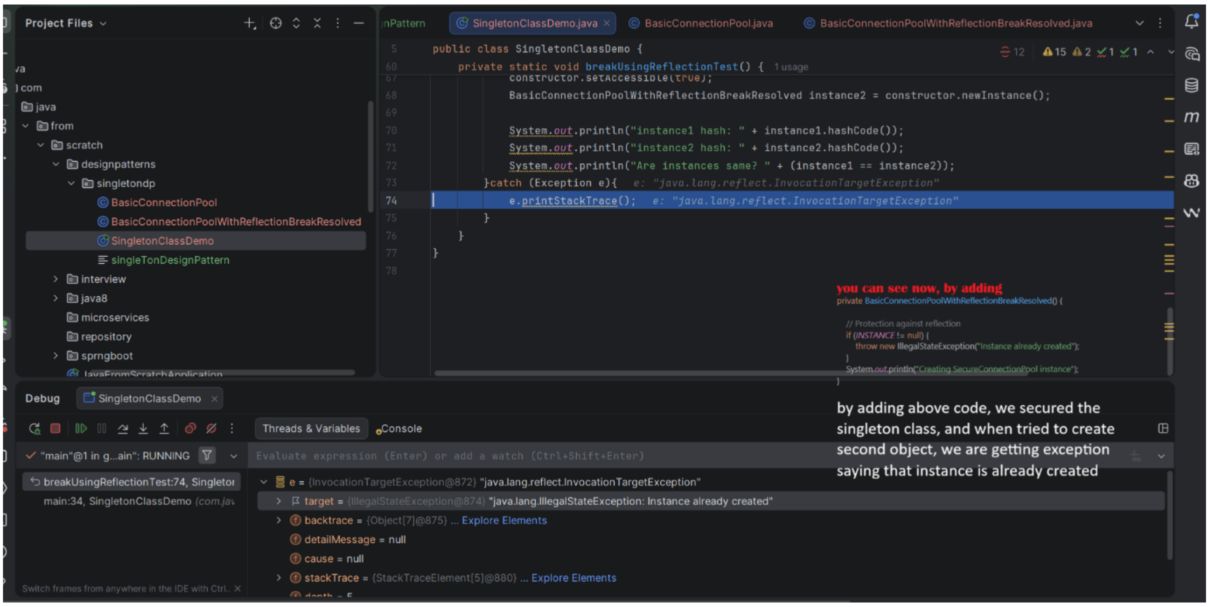
```java
public class SingletonClassDemo {
    }

    private static void breakUsingReflection() {  1 usage
        try {
            BasicConnectionPool instance1 = BasicConnectionPool.getInstance();   instance1: BasicConnectionPool@848

            // Using reflection to break singleton
            Constructor<BasicConnectionPool> constructor =   constructor: "private com.java.from.scratch.designpatte
                    BasicConnectionPool.class.getDeclaredConstructor();
            constructor.setAccessible(true);
            BasicConnectionPool instance2 = constructor.newInstance();   instance2: BasicConnectionPool@870    constr

            System.out.println("instance1 hash: " + instance1.hashCode());
            System.out.println("instance2 hash: " + instance2.hashCode());
            System.out.println("Are instances same? " + (instance1 == instance2 = false ));   instance2: BasicConnect
        }catch (Exception e){
            e.printStackTrace();
        }
```

```
"C:\Program Files\Eclipse Adoptium\jdk-21.0.9.10-hotspot\bin\java.exe" ...
Connected to the target VM, address: '127.0.0.1:58155', transport: 'socket'
Creating BasicConnectionPool instance
Creating BasicConnectionPool instance
instance1 hash: 691163666
instance2 hash: 2026877666
```

as we can see we have breaked eager singleton class using the reflection api and we can see the 2 different instances are created , since the hashcode of 2 instances are different

Why This Works

1. The INSTANCE is final and initialized when the class is loaded
2. The constructor checks if INSTANCE is not null before allowing creation
3. Since INSTANCE is final and set during class loading, it will never be null when the constructor runs for the first time
4. Any subsequent attempts to create an instance (including via reflection) will fail because INSTANCE is already set



```java
public class SingletonClassDemo {
    private static void breakUsingReflectionTest() {  1 usage
            constructor.setAccessible(true);
            BasicConnectionPoolWithReflectionBreakResolved instance2 = constructor.newInstance();

            System.out.println("instance1 hash: " + instance1.hashCode());
            System.out.println("instance2 hash: " + instance2.hashCode());
            System.out.println("Are instances same? " + (instance1 == instance2));
        }catch (Exception e){   e: "java.lang.reflect.InvocationTargetException"
            e.printStackTrace();   e: "java.lang.reflect.InvocationTargetException"
        }
    }
}
```

you can see now, by adding

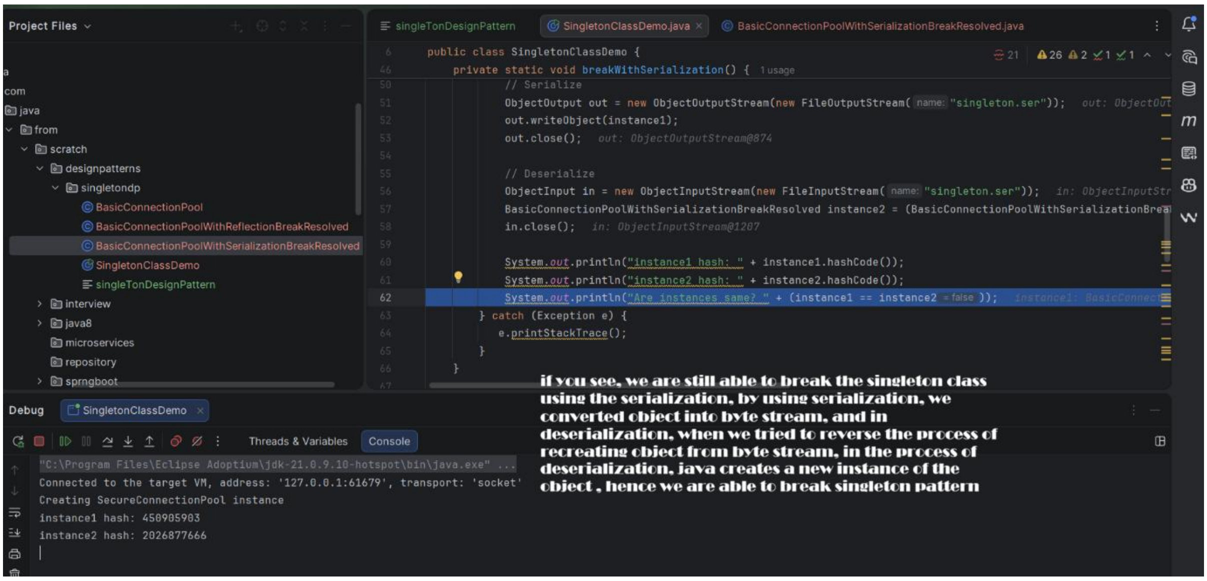
```java
private BasicConnectionPoolWithReflectionBreakResolved() {

  // Protection against reflection
  if (INSTANCE != null) {
    throw new IllegalStateException("Instance already created");
  }
  System.out.println("Creating SecureConnectionPool instance");
}
```

by adding above code, we secured the singleton class, and when tried to create second object, we are getting exception saying that instance is already created

```
"main"@1 in g...ain": RUNNING
breakUsingReflectionTest:74, Singletor
main:34, SingletonClassDemo (com.jav

Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
e = {InvocationTargetException@872} "java.lang.reflect.InvocationTargetException"
  target = {IllegalStateException@874} "java.lang.IllegalStateException: Instance already created"
  backtrace = {Object[7]@875} ... Explore Elements
  detailMessage = null
  cause = null
  stackTrace = {StackTraceElement[5]@880} ... Explore Elements

Switch frames from anywhere in the IDE with Ctrl..
```

We can still break it using serialization deserialization

- For this your singleton class must implements Serializable interface



```java
public class SingletonClassDemo {
    private static void breakWithSerialization() {
            // Serialize
            ObjectOutput out = new ObjectOutputStream(new FileOutputStream("singleton.ser"));
            out.writeObject(instance1);
            out.close();

            // Deserialize
            ObjectInput in = new ObjectInputStream(new FileInputStream("singleton.ser"));
            BasicConnectionPoolWithSerializationBreakResolved instance2 = (BasicConnectionPoolWithSerializationBrea
            in.close();

            System.out.println("instance1 hash: " + instance1.hashCode());
            System.out.println("instance2 hash: " + instance2.hashCode());
            System.out.println("Are instances same? " + (instance1 == instance2 ));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
"C:\Program Files\Eclipse Adoptium\jdk-21.0.9.10-hotspot\bin\java.exe" ...
Connected to the target VM, address: '127.0.0.1:61679', transport: 'socket'
Creating SecureConnectionPool instance
instance1 hash: 450905903
instance2 hash: 2026877666
```

if you see, we are still able to break the singleton class using the serialization, by using serialization, we converted object into byte stream, and in deserialization, when we tried to reverse the process of recreating object from byte stream, in the process of deserialization, java creates a new instance of the object , hence we are able to break singleton pattern

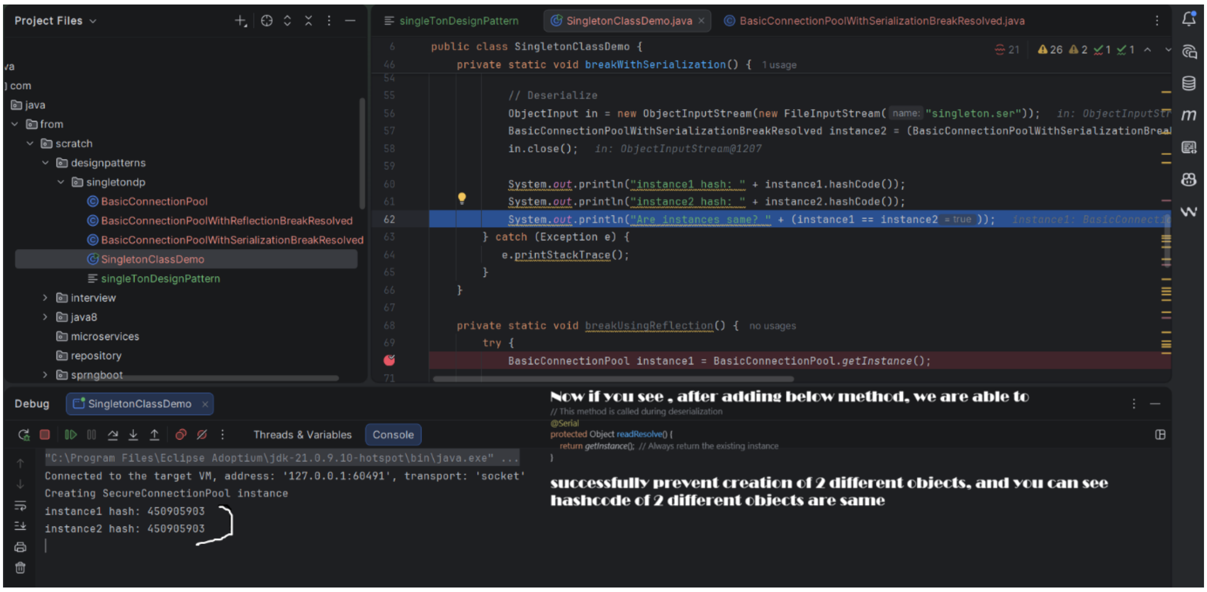
The Problem

1. **Serialization** is the process of converting an object into a byte stream.
2. **Deserialization** is the reverse process of recreating the object from the byte stream.
3. The issue is that during deserialization, Java creates a new instance of the object, which violates the singleton pattern.

Solution to serialization breaking

- We need to add below method to the singleton class

```java
// This method is called during deserialization
@Serial
protected Object readResolve() {
    return getInstance();   // Always return the existing instance
}
```



```
public class SingletonClassDemo {
    private static void breakWithSerialization() {  1 usage

            // Deserialize
            ObjectInput in = new ObjectInputStream(new FileInputStream( name: "singleton.ser"));   in: ObjectInputStr
            BasicConnectionPoolWithSerializationBreakResolved instance2 = (BasicConnectionPoolWithSerializationBrea
            in.close();   in: ObjectInputStream@1207

            System.out.println("instance1 hash: " + instance1.hashCode());
            System.out.println("instance2 hash: " + instance2.hashCode());
            System.out.println("Are instances same? " + (instance1 == instance2 = true ));   instance1: BasicConnecti
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void breakUsingReflection() {  no usages
        try {
            BasicConnectionPool instance1 = BasicConnectionPool.getInstance();
```

```
"C:\Program Files\Eclipse Adoptium\jdk-21.0.9.10-hotspot\bin\java.exe" ...
Connected to the target VM, address: '127.0.0.1:60491', transport: 'socket'
Creating SecureConnectionPool instance
instance1 hash: 450905903
instance2 hash: 450905903
```

Now if you see , after adding below method, we are able to

```
// This method is called during deserialization
@Serial
protected Object readResolve() {
    return getInstance(); // Always return the existing instance
}
```

successfully prevent creation of 2 different objects, and you can see hashcode of 2 different objects are same

Since above is eager initialization case, our singleton class is basically safe even if we use it in multi-threaded env, since instance is already created

Because:

- It uses static initialization, which is thread-safe
- The instance is final and created when the class is loaded
- There's no lazy initialization that could cause race conditions
-

Breaking singleTonClass with Lazy initialization (breaking in multi-threaded env)

Made below changes to the singleton class to make it lazy initialized

- Created instance like below, removed final keyword, because, we want to initialize it later in get instance method
- private static BasicConnectionPoolWithLazyInitialization INSTANCE;
- constructor added like below

```
public static BasicConnectionPoolWithLazyInitialization getInstance()
{
     if (INSTANCE == null) {
          INSTANCE = new BasicConnectionPoolWithLazyInitialization();
     }
     return INSTANCE;
}
```

Now lets see what are the problems in these Lazy initializations and how to resolve them

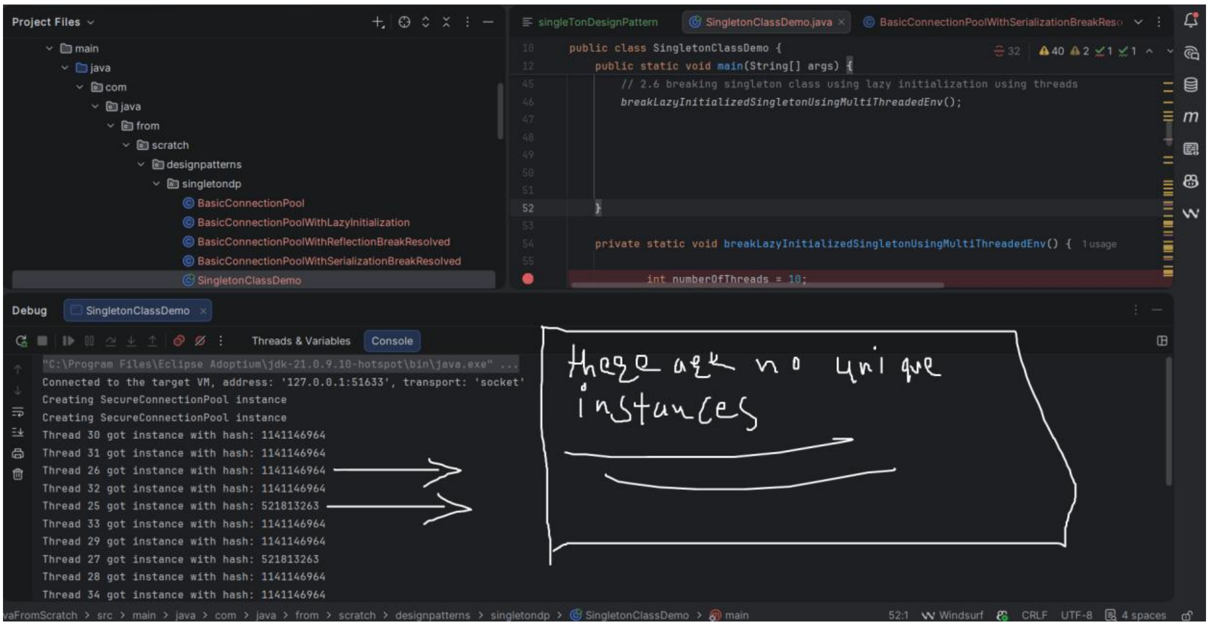
- since we have not added the synchronized blocks race condition can occur
- race condition can lead to multiple instances get created in a multi-threaded env
- Thread A enters get Instance check → it gets instance == null and gets true
- Thread B also enters get Instance() at the same time, checks INSTANCE == null and also gets true

- Thread A proceeds to create a new instance and assigns it to INSTANCE
- Thread B also proceeds to create another instance and assigns it to INSTANCE
- Now we have two different instances of the singleton class

How to Break It

Let's write a test to demonstrate this issue:

Method: *breakLazyInitializedSingletonUsingMultiThreadedEnv()*


Project Files
main
java
com
java
from
scratch
designpatterns
singletondp
BasicConnectionPool
BasicConnectionPoolWithLazyInitialization
BasicConnectionPoolWithReflectionBreakResolved
BasicConnectionPoolWithSerializationBreakResolved
SingletonClassDemo
singleTonDesignPattern
SingletonClassDemo.java
BasicConnectionPoolWithSerializationBreakReso
Debug
SingletonClassDemo
Threads & Variables
Console
vaFromScratch > src > main > java > com > java > from > scratch > designpatterns > singletondp > SingletonClassDemo > main
52:1 Windsurf CRLF UTF-8 4 spaces

```
public class SingletonClassDemo {
    public static void main(String[] args) {
        // 2.6 breaking singleton class using lazy initialization using threads
        breakLazyInitializedSingletonUsingMultiThreadedEnv();




    }

    private static void breakLazyInitializedSingletonUsingMultiThreadedEnv() {  1 usage

            int numberOfThreads = 10;
```

```
"C:\Program Files\Eclipse Adoptium\jdk-21.0.9.10-hotspot\bin\java.exe" ...
Connected to the target VM, address: '127.0.0.1:51633', transport: 'socket'
Creating SecureConnectionPool instance
Creating SecureConnectionPool instance
Thread 30 got instance with hash: 1141146964
Thread 31 got instance with hash: 1141146964
Thread 26 got instance with hash: 1141146964
Thread 32 got instance with hash: 1141146964
Thread 25 got instance with hash: 521813263
Thread 33 got instance with hash: 1141146964
Thread 29 got instance with hash: 1141146964
Thread 27 got instance with hash: 521813263
Thread 28 got instance with hash: 1141146964
Thread 34 got instance with hash: 1141146964
```

there are no unique instances

- In above screenshot, if you can see , there are more than 1 instance got created by different threads

How to prevent more than 1 instance creation in lazy initialization

- Make getInstance synchronized
- public static synchronized BasicConnectionPoolWithLazyInitialization getInstance() {
- Thread Safety: The synchronized keyword ensures that only one thread can execute this method at a time, preventing multiple threads from creating separate instances.
- Happens-Before Guarantee: The synchronized block provides a happens-before relationship, ensuring that all threads will see a fully constructed object.
- Memory Visibility: The synchronized block ensures that changes to the INSTANCE variable are visible to all threads.
- The first thread to acquire the lock will create the instance.
- All subsequent threads will see the non-null INSTANCE and return it immediately.
- The synchronized block prevents the race condition where multiple threads could pass the null check before any of them creates the instance.

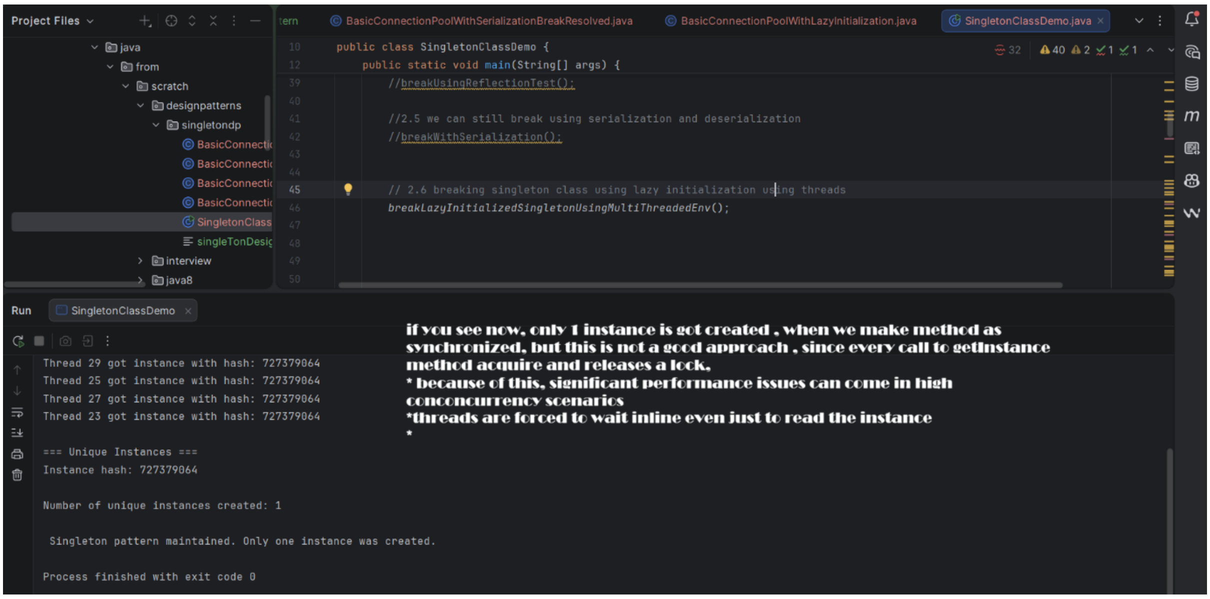
Pros:

- Simple to implement
- Thread-safe

Cons:

- Performance overhead on every method call due to synchronization

Testing it:



```java
public class SingletonClassDemo {
    public static void main(String[] args) {
        //breakUsingReflectionTest();

        //2.5 we can still break using serialization and deserialization
        //breakWithSerialization();


        // 2.6 breaking singleton class using lazy initialization using threads
        breakLazyInitializedSingletonUsingMultiThreadedEnv();
```

```
Thread 29 got instance with hash: 727379064
Thread 25 got instance with hash: 727379064
Thread 27 got instance with hash: 727379064
Thread 23 got instance with hash: 727379064

=== Unique Instances ===
Instance hash: 727379064

Number of unique instances created: 1

 Singleton pattern maintained. Only one instance was created.

Process finished with exit code 0
```

if you see now, only 1 instance is got created , when we make method as synchronized, but this is not a good approach , since every call to getInstance method acquire and releases a lock,
* because of this, significant performance issues can come in high conconcurrency scenarios
*threads are forced to wait inline even just to read the instance
*

Solution to not making whole method synchronized

Fina and recommended Solution: **Bill Pugh Singleton (Recommended)**

- Step 1: Remove the static INSTANCE variable
- // Remove this line:
- // private static BasicConnectionPoolWithLazyInitialization INSTANCE;
- Step 2: Remove the isInstanceCreated flag (not needed)
- // Remove this line:
- // private static boolean isInstanceCreated = false;
- Step 3: Add the static inner Holder class
- private static class Holder {
-     static final BasicConnectionPoolWithLazyInitialization INSTANCE =
-         new BasicConnectionPoolWithLazyInitialization();
- }
- Step 4: Update the getInstance() method

```
-   public static BasicConnectionPoolWithLazyInitialization
    getInstance() {
-       return Holder.INSTANCE;
-   }
-   Step 5: Update the constructor check
-   private BasicConnectionPoolWithLazyInitialization() {
-       // Protection against reflection
-       if (Holder.INSTANCE != null) {
-           throw new IllegalStateException("Use getInstance() method
    to get the single instance of this class.");
-       }
-       System.out.println("Creating SecureConnectionPool
    instance");
-   }
-
-
```

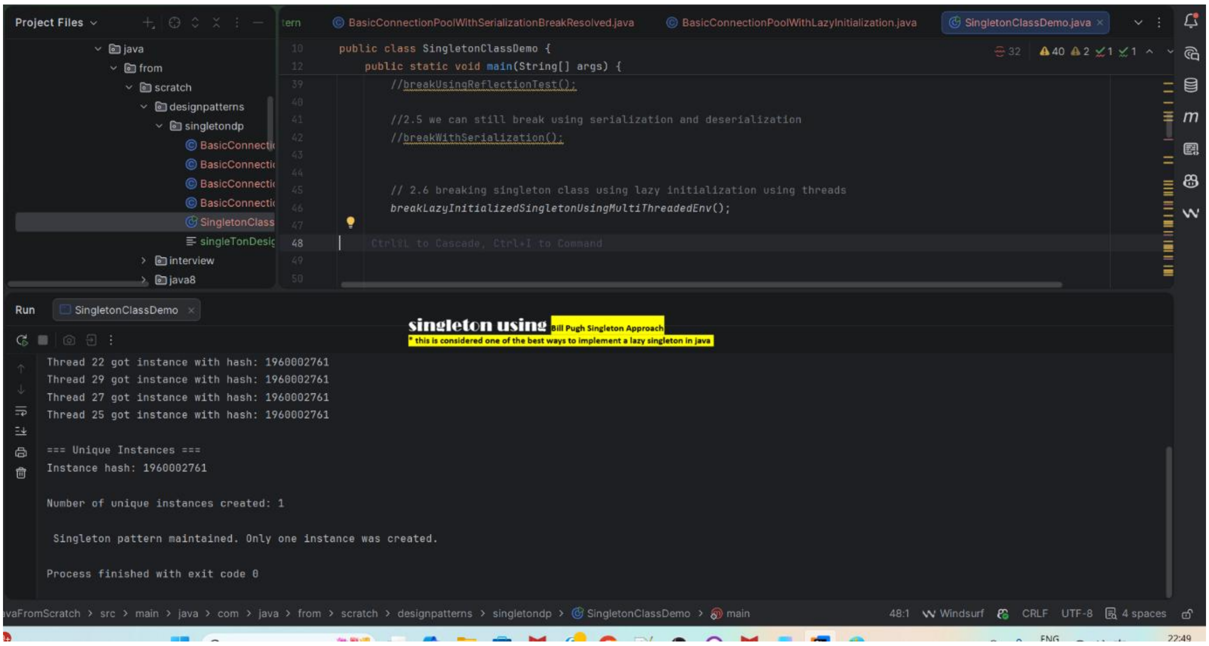
Key Benefits of Bill Pugh Singleton Approach:

1. **Thread Safety: Guaranteed by the JVM's class loading mechanism.**
2. **Lazy Initialization: The instance is created only when getInstance() is first called.**
3. **No Synchronization Overhead: No need for synchronized keyword.**
4. **Clean and Simple: Easy to understand and maintain.**
5. **Reflection Protection: The constructor throws an exception if someone tries to create a second instance through reflection.**

How It Works:

1. **The Holder class is not loaded into memory until getInstance() is called.**
2. **When getInstance() is first called, the JVM loads the Holder class and creates the INSTANCE.**
3. **The JVM guarantees that class loading is thread-safe, so no additional synchronization is needed.**
4. **Subsequent calls to getInstance() return the same instance.**

This is considered one of the best ways to implement a singleton in Java.



```java
public class SingletonClassDemo {
    public static void main(String[] args) {
        //breakUsingReflectionTest();

        //2.5 we can still break using serialization and deserialization
        //breakWithSerialization();


        // 2.6 breaking singleton class using lazy initialization using threads
        breakLazyInitializedSingletonUsingMultiThreadedEnv();

        Ctrl⇧L to Cascade, Ctrl+I to Command
```

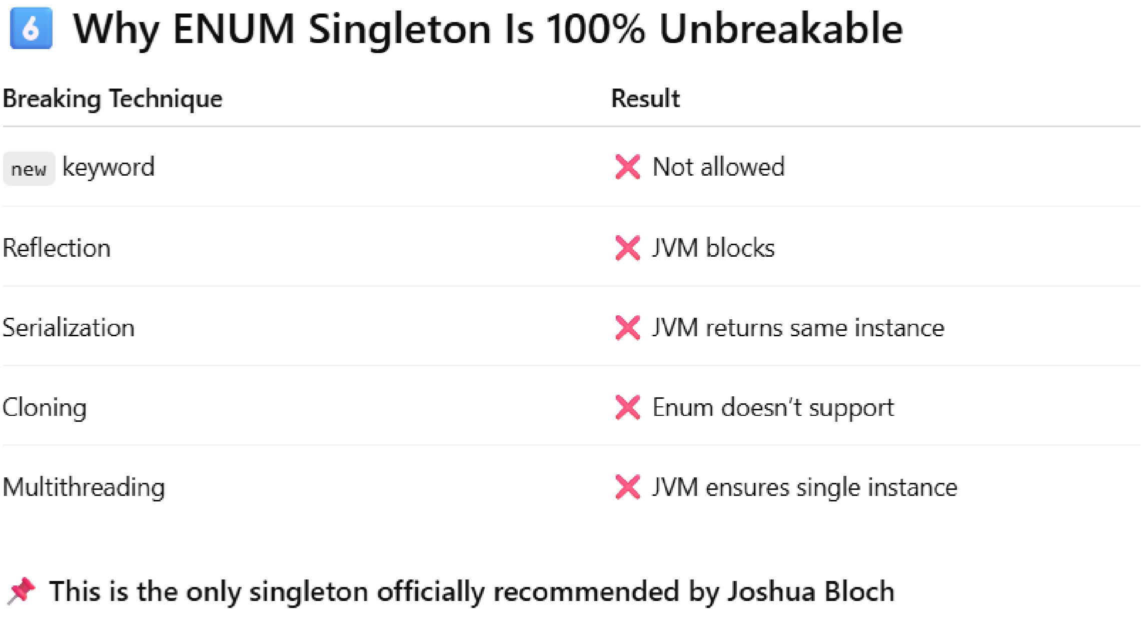
singleton using Bill Pugh Singleton Approach

* this is considered one of the best ways to implement a lazy singleton in java

```
Thread 22 got instance with hash: 1960002761
Thread 29 got instance with hash: 1960002761
Thread 27 got instance with hash: 1960002761
Thread 25 got instance with hash: 1960002761

=== Unique Instances ===
Instance hash: 1960002761

Number of unique instances created: 1

 Singleton pattern maintained. Only one instance was created.

Process finished with exit code 0
```

vaFromScratch > src > main > java > com > java > from > scratch > designpatterns > singletondp > SingletonClassDemo > main
48:1
Windsurf
CRLF
UTF-8
4 spaces
ENG
22:49

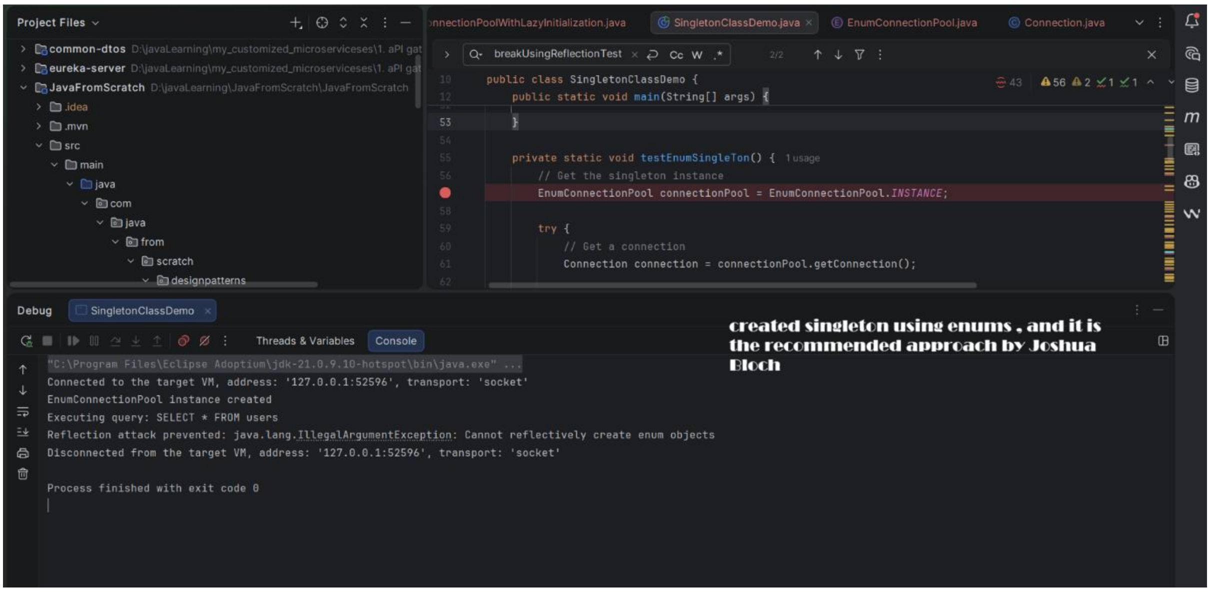
But the very much recommended approach is Enum singleton



6 Why ENUM Singleton Is 100% Unbreakable

Breaking Technique	Result
`new` keyword	❌ Not allowed
Reflection	❌ JVM blocks
Serialization	❌ JVM returns same instance
Cloning	❌ Enum doesn’t support
Multithreading	❌ JVM ensures single instance

📌 This is the only singleton officially recommended by Joshua Bloch


Project Files
common-dtos D:\javaLearning\my_customized_microservices\1. aPI gat
eureka-server D:\javaLearning\my_customized_microservices\1. aPI gat
JavaFromScratch D:\javaLearning\JavaFromScratch\JavaFromScratch
.idea
.mvn
src
main
java
com
java
from
scratch
designpatterns
onnectionPoolWithLazyInitialization.java
SingletonClassDemo.java
EnumConnectionPool.java
Connection.java
breakUsingReflectionTest
2/2
Debug
SingletonClassDemo
Threads & Variables
Console

```
public class SingletonClassDemo {
    public static void main(String[] args) {

    }

    private static void testEnumSingleTon() {  1 usage
        // Get the singleton instance
        EnumConnectionPool connectionPool = EnumConnectionPool.INSTANCE;

        try {
            // Get a connection
            Connection connection = connectionPool.getConnection();
```

```
"C:\Program Files\Eclipse Adoptium\jdk-21.0.9.10-hotspot\bin\java.exe" ...
Connected to the target VM, address: '127.0.0.1:52596', transport: 'socket'
EnumConnectionPool instance created
Executing query: SELECT * FROM users
Reflection attack prevented: java.lang.IllegalArgumentException: Cannot reflectively create enum objects
Disconnected from the target VM, address: '127.0.0.1:52596', transport: 'socket'

Process finished with exit code 0
```

created singleton using enums , and it is the recommended approach by Joshua Bloch

Why Enum Singleton is the Best:

Thread Safety: Enum instances are thread-safe by default.

Serialization: Handled automatically by the JVM.

Reflection Attack Protection: Java ensures enum instances are instantiated only once, even with reflection.

Simple and Clean: Less code, more readable.

Lazy Initialization: The instance is created when the enum is first accessed.

Real-World Usage:

Database Connection Pools: HikariCP, C3P0, etc.

Logging Frameworks: Log4j, SLF4J

Configuration Management: Loading application config once

Caching Mechanisms: EhCache, Guava Cache

Thread Pools: ExecutorService instances