**Singleton**

* **Introduction:**
  + A singleton class has only one instance, accessible globally through a single point (via a method/field).
  + Main problem this pattern solves is to ensure that only a single instance of this class exists.
  + Any state you add in your singleton becomes part of “global state” of your application.
* **UML:**

A close up of a device

Description automatically generated

* **Implementation steps:**
  + Controlling instance creation
    - Class constructor(s) must not be accessible globally.
    - Subclassing / inheritance must not be allowed.
  + Keeping track of instance
    - Class itself a good place to track the instance.
  + Giving access to the singleton instance
    - A public static method is a good choice.
    - Can expose instances as final public static field but it won’t work for all singleton instances.
  + Two options for implementing a singleton
    - Early initialization – Eager Singleton
      * Create singleton as soon as the class is loaded.
    - Lazy initialization – Lazy singleton
      * Singleton is created when it is first requested.
* **Implementation & Design Consideration:**
  + Early / Eager initialization is the simplest and preferred way. Always try to use this approach first.
  + The “classic” singleton pattern implementations double check locking and volatile field.
  + The lazy initialization holder idiom provides best of both worlds, you don’t deal with synchronization issues directly and is easy to implement.
  + You can also implement singleton using enums. However due to pre-conceptions about what an enum is, it may be a hard sell during code review especially if singleton has mutable fields.
  + If the simple solution works then use it.

**Design Consideration:**

Singleton creation doesn’t need any parameters. If you find yourself in need of support for constructor arguments, you need a simple factory or factory method pattern instead.

Make sure that your singletons are not carrying a lot of mutable global state.

* **Example:**

The java.lang.Runtime class is standard Java API is a singleton.

A screenshot of a cell phone

Description automatically generated

Logging frameworks also makes use of Singleton pattern.

Spring framework treats all beans by default as Singleton. In spring we don’t have to make any changes to ensure single instance, Spring handles that for us.

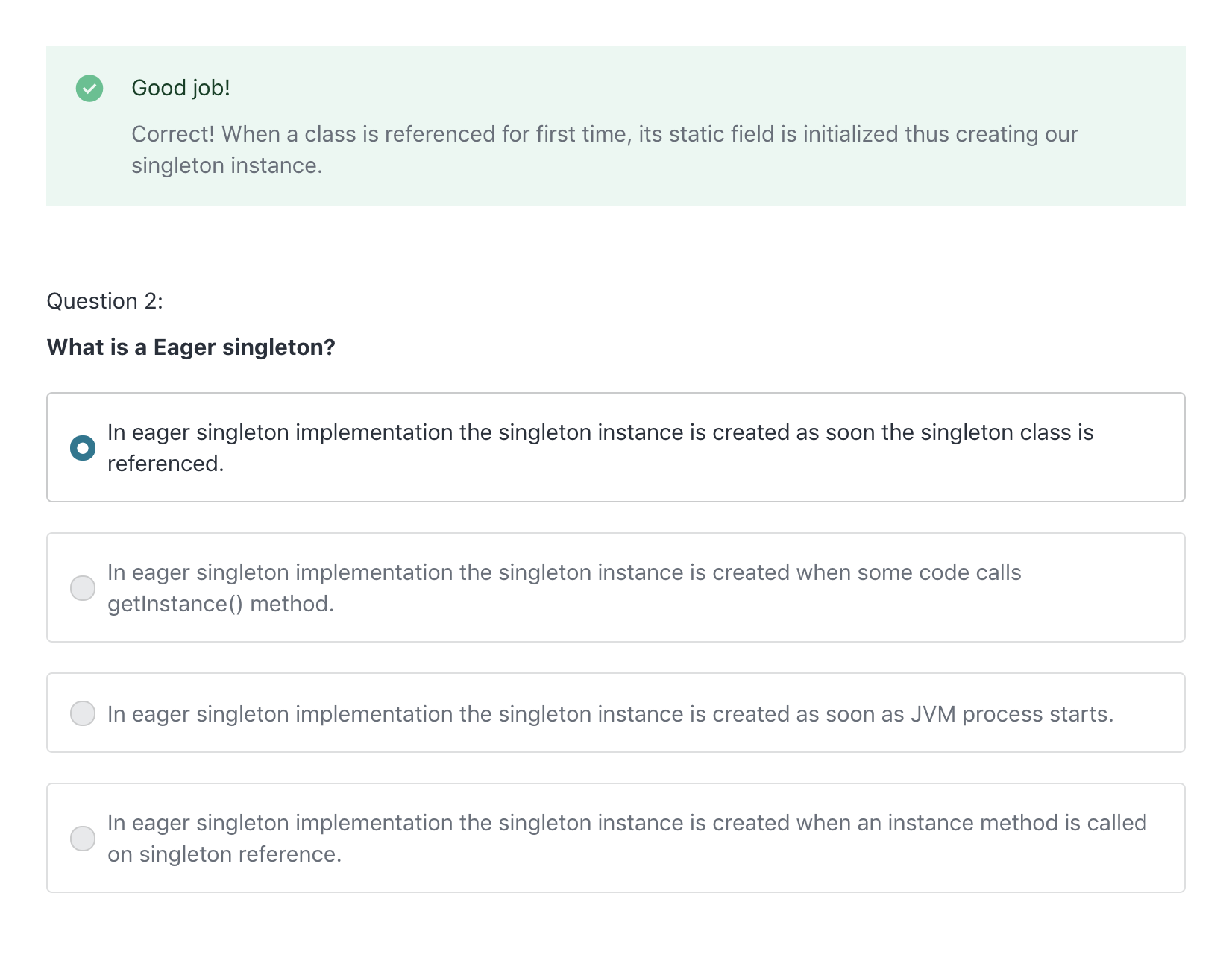
* **Comparison with Factory Method:**

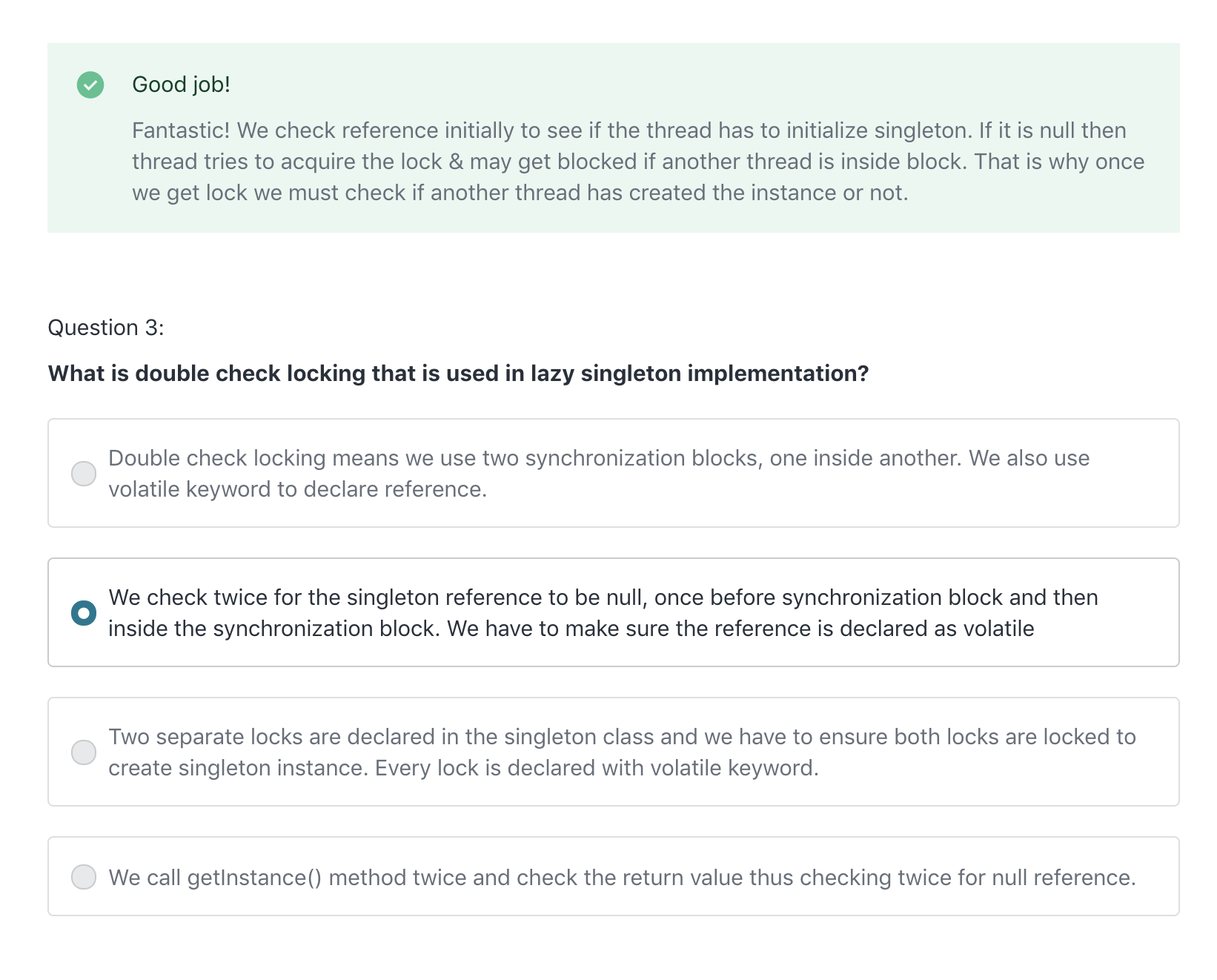
|  |  |
| --- | --- |
| **Singleton** | **Factory Method** |
| Primary purpose or intent of singleton pattern is to ensure that only one instance of a class is ever created. | Factory Method is primarily used to isolate client code from object creation & delegate object creation to subclass. |
| Singleton instance is created without any need of arguments from client code. | Factory method allows to parameterize the object creation. |

* **Pitfalls:**
* Singleton pattern can deceive you about true dependencies! Since they are globally accessible its easy to miss dependencies.
* They are hard to unit test. You cannot easily mock the instance that is returned.
* Most common way to implement singleton in Java is through static variable and they are held per class loader and not per JVM. So they may not be truly Singleton in an OSGI or web application.
* A Singleton carrying around a large mutable global state is a good indication of an abused Singleton pattern.
* **Quiz:**

**A screenshot of a social media post

Description automatically generated**

****

****

**A screenshot of a social media post

Description automatically generated**