C#

The C# programming language is based on Microsoft .NET, the popular platform in the Windows ecosystem. Many companies focusing on Windows development include requirements on C# skills in their job descriptions.

C# can be viewed as a managed programming language based on C++, so there are many similarities between them. It does not take much time for a C++ programmer to learn C#. It is easy to learn the similarities, but it is difficult to distinguish differences. Many interviewers like to ask questions about the confusing differences. For example, the following is a piece of dialog from an interview:

Interviewer: Types in C++ can be defined as `struct` and `class`. What are the differences between these two types?

Candidate: The default access level in a `struct` is public, while in a `class` is private.

Interviewer: How about in C#?

Candidate: It is different in C#. The default access level for both `struct` and `class` is private in C#. However, their memory allocation models are different. A `struct` is a value type, and its instances are created on the stack. A type declared as a `class` is a reference type, and its instances are created on the heap.

Similar to C++, every type in C# has a constructor at least. Additionally, there might be two different methods (finalizer and `Dispose`) used to release resources in C#. The finalizer method looks similar to the destructor in C++, which is the type name followed by a complement symbol (~). However, the time to invoke them is different. The time to invoke a destructor in C++ is fixed, but the time to invoke the finalizer in C# is determined by the runtime. The finalizer is invoked by the garbage collector, and the time is unknown to programmers.

Additionally, there is a special constructor in C#, named static constructor. It is invoked automatically when its type is executed by the runtime at the first time, and it is invoked only once. There are many interesting interview questions about the static constructor. For example: What is the output of the C# code in Listing 2-12?

Listing 2-12. *C# Code about Static Constructors*

```
class A {
    public A(string text) {
        Console.WriteLine(text);
    }
}

class B{
    static A a1 = new A("a1");
    A a2 = new A("a2");

       static B() {
        a1 = new A("a3");
    }

    public B() {
        a2 = new A("a4");
    }
}

class Program {
    static void Main(string[] args) {
        B b = new B();
    }
}
```

Before any code of the type `B` gets executed, its static constructor will be invoked first. It initializes static data fields and then executes statements inside the static constructor. Therefore, it prints a line of "a1", and then another line of "a3".

When the execution flow reaches the statement `B b = new B()`, the ordinary constructor is invoked. It initializes its data fields first and then the statements in its constructor method, so it prints a line of "a2", and then another line of "a4".

Singleton

Question 3 Please design and implement a class of which we can only create a single instance.

A class follows the singleton pattern if it can have one instance at most. Design patterns are very important in object-oriented design and development, so many

interviewers like questions related to patterns. Singleton is the only pattern that can be implemented in dozens of lines of code, so it is quite suitable for interviews.

Workable Only in Single-Threading Applications

When the constructor is defined as a private method, none of the code outside the class can create its instances. A static method inside the class is defined to create an instance on demand. The class `Singleton1` is implemented based on the solution in Listing 2-13.

Listing 2-13. *C# Code for Singleton (Version 1)*

```
public class Singleton1 {

 private Singleton1() {

 }


    private static Singleton1 instance = null;

 public static Singleton1 Instance {

     get {

         if (instance == null)

             instance = new Singleton1();

         return instance;

     }

 }

}
```

In this class, an instance is created only when static field `Singleton1.instance` is `null`, so it does not have the opportunity to get multiple instances.

Works with Multiple Threads but Is Inefficient

`Singleton1` works when there is only one thread, but it has problems when there are multiple threads in an application. Supposing that there are two threads concurrently reaching the `if` statement to check whether `instance` is `null`. If `instance` is not created yet, each thread will create one separately. It violates the definition of the singleton pattern when two instances are created. In order to make it work in multithreading applications, a lock is introduced as shown in the `Singleton2` in Listing 2-14.

Listing 2-14. *C# Code for Singleton (Version 2)*

```
public class Singleton2 {

 private Singleton2() {

 }


 private static readonly object syncObj = new object();


 private static Singleton2 instance = null;

 public static Singleton2 Instance {

     get {

         lock (syncObj) {

             if (instance == null)

                 instance = new Singleton2();

         }


         return instance;

     }

 }

}
```

Suppose there are two threads that are both going to create their own instances. As we know, only one thread can get the lock at a time. When one thread gets it, the other one has to wait. The first thread that gets the lock finds that `instance` is `null`, so it creates an instance. After the first thread releases the lock, the second thread gets it. Since the `instance` was already created by the first thread, the `if` statement is `false`. An instance will not be recreated again. Therefore, it guarantees that there is one instance at most when there are multiple threads executing concurrently.

The class `Singleton2` is far from perfect. Every time `Singleton2.Instance.get` executes, it has to get and release a lock. Operations to get and release a lock are time-consuming, so they should be avoided as much as possible.

Double-Check around Locking

Actually a lock is needed only before the only instance is created in order to make sure that only one thread get the chance to create an instance. After the instance is created, no lock is necessary. We can improve performance with an additional `if` check before the lock as shown in Listing 2-15.

Listing 2-15. *C# Code for Singleton (Version 3)*

```
public class Singleton3 {
```

```
    private Singleton3() {
    }

    private static object syncObj = new object();

    private static Singleton3 instance = null;
    public static Singleton3 Instance {
        get {
            if (instance == null) {
                lock (syncObj) {
                    if (instance == null)
                        instance = new Singleton3();
                }
            }

            return instance;
        }
    }
}
```

In the class `Singleton3`, it locks only when `instance` is `null`. When the instance has been created, it is returned directly without any locking operations. Therefore, the time efficiency of `Singleton3` is better than `Singleton2`.

`Singleton3` employs two `if` statements to improve time efficiency. It is a workable solution, but its logic looks a bit complex, and it is error-prone for many candidates during interviews. Let us explore simpler, and also better, solutions.

Utilization of Static Constructors

It is guaranteed that a static constructor in a C# class is called only once at most. If it only creates an instance in the static constructor, there is one instance at most. A concise solution for the singleton pattern with a static constructor is shown in Listing 2-16.

Listing 2-16. *C# Code for Singleton (Version 4)*

```
public class Singleton4 {
    private Singleton4() {
    }

    private static Singleton4 instance = new Singleton4();
    public static Singleton4 Instance {
        get {
            return instance;
        }
    }
}
```

In the class `Singleton4` above, an instance is created when the static field `instance` gets initialized. Static fields in C# are initialized when the static constructor is called. Since the static constructor is called only once by the .NET runtime, it is guaranteed that only one instance is created even in a multithreading application.

The time to execute the static constructor is out of the programmers' control. When the .NET runtime reaches any code of a class the first time, it invokes the static constructor automatically. Therefore, the time to initialize `instance` is not the first time to invoke `Singleton4.Instance`. If a static method is added into `Singleton4`, it is not necessary to create an instance to invoke such a static method. However, the .NET runtime invokes the static constructors automatically to create an instance when it reaches any code for `Singleton4`. Therefore, it is possible to create an instance too early, and it impairs the space efficiency.

Creating an Instance When Necessary

In the last implementation of the singleton pattern in Listing 2-17, `Singleton5`, it creates the only instance on demand.

Listing 2-17. *C# Code for Singleton (Version 5)*

```
public class Singleton5 {
    Singleton5() {
    }

    public static Singleton5 Instance {
```

```
        get {
            return Nested.instance;
        }
       }

    class Nested {
        static Nested() {
        }

        internal static readonly Singleton5 instance = new Singleton5();
    }
}
```

There is a nested private class `Nested` in the code for `Singleton5`. When the .NET runtime reaches the code of the class `Nested`, its static constructor is invoked automatically, which creates an instance of type `Singleton5`. The class `Nested` is used only in the property `Singleton5.Instance`. Since the nested class is defined as private, it is inaccessible outside of the class `Singleton5`.

When the `get` method of `Singleton5.Instance` is invoked the first time, it triggers execution of the static constructor of the class `Nested` to create an instance of `Singleton5`. The instance is created only when it is necessary, so it avoids the waste associated with creating the instance too early.

Solution Comparison

Five solutions are introduced in this section. The first solution is workable only in a single-threading application. The second one works in a multiple-threading application, but it is inefficient because of unnecessary locking operations. The first two solutions are not acceptable from an interviewer's perspective. In the third solution, it employs two `if` statements and one lock to make sure it works in a multiple-threading application efficiently. The fourth one utilizes the static constructor to guarantee only an instance is created. The last solution improves space efficiency with a nested class to create the instance only when it is necessary. The last two solutions are recommended for interviews.

Source Code:

`003_Singleton.cs`