Lazy<T> Class

Reference

Definition

Namespace: System

Assembly: mscorlib.dll

Provides support for lazy initialization.

In this article

C#

```
[System.Runtime.InteropServices.ComVisible(false)]
[System.Serializable]
public class Lazy<T>
```

Type Parameters

T

The type of object that is being lazily initialized.

Inheritance Object → Lazy<T>

Derived System.Lazy<T,TMetadata>

Attributes ComVisibleAttribute, SerializableAttribute

Examples

The following example demonstrates the use of the Lazy<T> class to provide lazy initialization with access from multiple threads.

> **Note**
>
> The example uses the **Lazy<T>(Func<T>)** constructor. It also demonstrates the use of the **Lazy<T>(Func<T>, Boolean)** constructor (specifying `true` for `isThreadSafe`) and the **Lazy<T>(Func<T>, LazyThreadSafetyMode)** constructor (specifying **LazyThreadSafetyMode.ExecutionAndPublication** for `mode`). To switch to a different constructor, just change which constructors are commented out.
>
> For an example that demonstrates exception caching using the same constructors, see the **Lazy<T>(Func<T>)** constructor.

The example defines a `LargeObject` class that will be initialized lazily by one of several threads. The four key sections of code illustrate the creation of the initializer, the factory method, the actual initialization, and the constructor of the `LargeObject` class, which displays a message when the object is created. At the beginning of the `Main` method, the example creates the thread-safe lazy initializer for `LargeObject`:

C#

```
lazyLargeObject = new Lazy<LargeObject>(InitLargeObject);

// The following lines show how to use other constructors to achieve exactly
the
// same result as the previous line:
//lazyLargeObject = new Lazy<LargeObject>(InitLargeObject, true);
//lazyLargeObject = new Lazy<LargeObject>(InitLargeObject,
//                                LazyThreadSafetyMode.ExecutionAndPublication);
```

The factory method shows the creation of the object, with a placeholder for further initialization:

C#

```
static LargeObject InitLargeObject()
{
    LargeObject large = new LargeObject(Thread.CurrentThread.ManagedThreadId);
```

```
    // Perform additional initialization here.
    return large;
}
```

Note that the first two code sections could be combined by using a lambda function, as shown here:

C#

```
lazyLargeObject = new Lazy<LargeObject>(() =>
{
    LargeObject large = new LargeObject(Thread.CurrentThread.ManagedThreadId);
    // Perform additional initialization here.
    return large;
});
```

The example pauses, to indicate that an indeterminate period may elapse before lazy initialization occurs. When you press the **Enter** key, the example creates and starts three threads. The `ThreadProc` method that's used by all three threads calls the Value property. The first time this happens, the `LargeObject` instance is created:

C#

```
LargeObject large = lazyLargeObject.Value;

// IMPORTANT: Lazy initialization is thread-safe, but it doesn't protect the
//            object after creation. You must lock the object before accessing
it,
//            unless the type is thread safe. (LargeObject is not thread safe.)
lock(large)
{
    large.Data[0] = Thread.CurrentThread.ManagedThreadId;
    Console.WriteLine("Initialized by thread {0}; last used by thread {1}.",
        large.InitializedBy, large.Data[0]);
}
```

The constructor of the `LargeObject` class, which includes the last key section of code, displays a message and records the identity of the initializing thread. The output from the program appears at the end of the full code listing.

C#

```
int initBy = 0;
public LargeObject(int initializedBy)
```

```
{
    initBy = initializedBy;
    Console.WriteLine("LargeObject was created on thread id {0}.", initBy);
}
```

> **Note**
>
> For simplicity, this example uses a global instance of `Lazy<T>`, and all the methods are `static` (`Shared` in Visual Basic). These are not requirements for the use of lazy initialization.

C#

```
using System;
using System.Threading;

class Program
{
    static Lazy<LargeObject> lazyLargeObject = null;

    static LargeObject InitLargeObject()
    {
        LargeObject large = new
LargeObject(Thread.CurrentThread.ManagedThreadId);
        // Perform additional initialization here.
        return large;
    }

    static void Main()
    {
        // The lazy initializer is created here. LargeObject is not created un-
til the
        // ThreadProc method executes.
        lazyLargeObject = new Lazy<LargeObject>(InitLargeObject);

        // The following lines show how to use other constructors to achieve
exactly the
        // same result as the previous line:
        //lazyLargeObject = new Lazy<LargeObject>(InitLargeObject, true);
        //lazyLargeObject = new Lazy<LargeObject>(InitLargeObject,
        //
LazyThreadSafetyMode.ExecutionAndPublication);

        Console.WriteLine(
            "\r\nLargeObject is not created until you access the Value property
of the lazy" +
            "\r\ninitializer. Press Enter to create LargeObject.");
```

```
        Console.ReadLine();

        // Create and start 3 threads, each of which uses LargeObject.
        Thread[] threads = new Thread[3];
        for (int i = 0; i < 3; i++)
        {
            threads[i] = new Thread(ThreadProc);
            threads[i].Start();
        }

        // Wait for all 3 threads to finish.
        foreach (Thread t in threads)
        {
            t.Join();
        }

        Console.WriteLine("\r\nPress Enter to end the program");
        Console.ReadLine();
    }

    static void ThreadProc(object state)
    {
        LargeObject large = lazyLargeObject.Value;

        // IMPORTANT: Lazy initialization is thread-safe, but it doesn't pro-
tect the
        //            object after creation. You must lock the object before
accessing it,
        //            unless the type is thread safe. (LargeObject is not
thread safe.)
        lock(large)
        {
            large.Data[0] = Thread.CurrentThread.ManagedThreadId;
            Console.WriteLine("Initialized by thread {0}; last used by thread
{1}.",
                large.InitializedBy, large.Data[0]);
        }
    }
}

class LargeObject
{
    public int InitializedBy { get { return initBy; } }

    int initBy = 0;
    public LargeObject(int initializedBy)
    {
        initBy = initializedBy;
        Console.WriteLine("LargeObject was created on thread id {0}.", initBy);
    }
```

```
    public long[] Data = new long[100000000];
}

/* This example produces output similar to the following:

LargeObject is not created until you access the Value property of the lazy
initializer. Press Enter to create LargeObject.

LargeObject was created on thread id 3.
Initialized by thread 3; last used by thread 3.
Initialized by thread 3; last used by thread 4.
Initialized by thread 3; last used by thread 5.

Press Enter to end the program
 */
```

Remarks

Use lazy initialization to defer the creation of a large or resource-intensive object, or the execution of a resource-intensive task, particularly when such creation or execution might not occur during the lifetime of the program.

To prepare for lazy initialization, you create an instance of Lazy<T>. The type argument of the Lazy<T> object that you create specifies the type of the object that you want to initialize lazily. The constructor that you use to create the Lazy<T> object determines the characteristics of the initialization. Lazy initialization occurs the first time the Lazy<T>.Value property is accessed.

In most cases, choosing a constructor depends on your answers to two questions:

- Will the lazily initialized object be accessed from more than one thread? If so, the Lazy<T> object might create it on any thread. You can use one of the simple constructors whose default behavior is to create a thread-safe Lazy<T> object, so that only one instance of the lazily instantiated object is created no matter how many threads try to access it. To create a Lazy<T> object that is not thread safe, you must use a constructor that enables you to specify no thread safety.

  > **⊗ Caution**
  >
  > Making the **Lazy<T>** object thread safe does not protect the lazily initialized object. If multiple threads can access the lazily initialized object, you must make its properties and methods safe for multithreaded access.

- Does lazy initialization require a lot of code, or does the lazily initialized object have a parameterless constructor that does everything you need and doesn't throw exceptions? If you need to write initialization code or if exceptions need to be handled, use one of the constructors that takes a factory method. Write your initialization code in the factory method.

The following table shows which constructor to choose, based on these two factors:

Object will be accessed by	If no initialization code is required (parameterless constructor), use	If initialization code is required, use
Multiple threads	Lazy<T>()	Lazy<T>(Func<T>)
One thread	Lazy<T>(Boolean) with `isThreadSafe` set to `false`.	Lazy<T>(Func<T>, Boolean) with `isThreadSafe` set to `false`.

You can use a lambda expression to specify the factory method. This keeps all the initialization code in one place. The lambda expression captures the context, including any arguments you pass to the lazily initialized object's constructor.

Exception caching When you use factory methods, exceptions are cached. That is, if the factory method throws an exception the first time a thread tries to access the Value property of the Lazy<T> object, the same exception is thrown on every subsequent attempt. This ensures that every call to the Value property produces the same result and avoids subtle errors that might arise if different threads get different results. The Lazy<T> stands in for an actual `T` that otherwise would have been initialized at some earlier point, usually during startup. A failure at that earlier point is usually fatal. If there is a potential for a recoverable failure, we recommend that you build the retry logic into the initialization routine (in this case, the factory method), just as you would if you weren't using lazy initialization.

Alternative to locking In certain situations, you might want to avoid the overhead of the Lazy<T> object's default locking behavior. In rare situations, there might be a potential for deadlocks. In such cases, you can use the Lazy<T>(LazyThreadSafetyMode) or Lazy<T>(Func<T>, LazyThreadSafetyMode) constructor, and specify LazyThreadSafetyMode.PublicationOnly. This enables the Lazy<T> object to create a copy of the lazily initialized object on each of several threads if the threads call the Value property simultaneously. The Lazy<T> object ensures that all threads use the same instance of the lazily initialized object and discards the instances that are not used. Thus,

the cost of reducing the locking overhead is that your program might sometimes create and discard extra copies of an expensive object. In most cases, this is unlikely. The examples for the Lazy<T>(LazyThreadSafetyMode) and Lazy<T>(Func<T>, LazyThreadSafetyMode) constructors demonstrate this behavior.

> **Important**
>
> When you specify **LazyThreadSafetyMode.PublicationOnly**, exceptions are never cached, even if you specify a factory method.

Equivalent constructors In addition to enabling the use of LazyThreadSafetyMode.PublicationOnly, the Lazy<T>(LazyThreadSafetyMode) and Lazy<T>(Func<T>, LazyThreadSafetyMode) constructors can duplicate the functionality of the other constructors. The following table shows the parameter values that produce equivalent behavior.

To create a Lazy<T> object that is	For constructors that have a `LazyThreadSafetyMode mode` parameter, set `mode` to	For constructors that have a Boolean `isThreadSafe` parameter, set `isThreadSafe` to	For constructors with no thread safety parameters
Fully thread safe; uses locking to ensure that only one thread initializes the value.	ExecutionAndPublication	`true`	All such constructors are fully thread safe.
Not thread safe.	None	`false`	Not applicable.
Fully thread safe; threads race to initialize the value.	PublicationOnly	Not applicable.	Not applicable.

Other capabilities For information about the use of Lazy<T> with thread-static fields, or as the backing store for properties, see Lazy Initialization.

Constructors

Lazy<T>()	Initializes a new instance of the Lazy<T> class. When lazy initialization occurs, the parameterless constructor of the target type is used.

Lazy<T>(Boolean)	Initializes a new instance of the Lazy<T> class. When lazy initialization occurs, the parameterless constructor of the target type and the specified initialization mode are used.
Lazy<T>(Func<T>)	Initializes a new instance of the Lazy<T> class. When lazy initialization occurs, the specified initialization function is used.
Lazy<T>(Func<T>, Boolean)	Initializes a new instance of the Lazy<T> class. When lazy initialization occurs, the specified initialization function and initialization mode are used.
Lazy<T>(Func<T>, LazyThreadSafetyMode)	Initializes a new instance of the Lazy<T> class that uses the specified initialization function and thread-safety mode.
Lazy<T>(LazyThreadSafetyMode)	Initializes a new instance of the Lazy<T> class that uses the parameterless constructor of `T` and the specified thread-safety mode.

Properties

IsValueCreated	Gets a value that indicates whether a value has been created for this Lazy<T> instance.
Value	Gets the lazily initialized value of the current Lazy<T> instance.

Methods

Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetType()	Gets the Type of the current instance. (Inherited from Object)
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)
ToString()	Creates and returns a string representation of the Value property for this instance.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0
Xamarin.iOS	10.8
Xamarin.Mac	3.0

Thread Safety

By default, all public and protected members of the Lazy<T> class are thread safe and may be used concurrently from multiple threads. These thread-safety guarantees may be removed optionally and per instance, using parameters to the type's constructors.

See also

- LazyThreadSafetyMode
- Lazy Initialization
- Lazy Expressions (F#)