**Readonly** prevents fields from being changed. Readonly fields can be initialized at runtime, unlike const values. Attempts to change them later are disallowed. This C# modifier does not affect performance.

**Const** is a keyword that indicates a constant. It describes an entity that cannot be changed at program runtime. Instead, the entity must be fully resolved at compile-time.  
**We cannot reassign** a constant. With constants, we lose the ability to modify variables at runtime but we gain performance and compile-time validation.

**Many locations** exist in computer memory. A static thing stands alone. It occupies just one location. Static is used on methods, classes and constructors.

**Static methods.** These are called with the type name. No instance is required—this makes them slightly faster. Static methods can be public or private.

**Info:**Static methods use the static keyword, usually as the first keyword or the second keyword after public.

**Warning:**A static method cannot access non-static class level members. It has no "this" pointer.

**Lazy** instantiation delays certain tasks. It typically improves the startup time of a C# application. This has always been possible to implement. But the .NET Framework now offers the Lazy type, which provides this feature.  
**To start,** let's examine a simple example of the Lazy type. In this program, you will first see the Test class. This class contains a constructor and an allocation occurs in the constructor.[**Constructor**](https://www.dotnetperls.com/constructor)  
**In the Main entry point,** a new Lazy instance of the Test type is created. The IsValueCreated property is False at this point. Then, when the Value property is accessed, the Test class constructor is executed and memory allocation occurs.

**Finally:**The IsValueCreated property returns True and the Test instance can be used like any other object.

[**Property**](https://www.dotnetperls.com/property)

**Based on:** .NET 4.5

**C# program that uses Lazy type**

using System;

class Test

{

int[] \_array;

public Test()

{

Console.WriteLine("Test()");

\_array = new int[10];

}

public int Length

{

get

{

return \_array.Length;

}

}

}

class Program

{

static void Main()

{

// Create Lazy.

**Lazy**<Test> lazy = new Lazy<Test>();

// Show that IsValueCreated is false.

Console.WriteLine("IsValueCreated = {0}", lazy.IsValueCreated);

// Get the Value.

// ... This executes Test().

Test test = lazy.Value;

// Show the IsValueCreated is true.

Console.WriteLine("IsValueCreated = {0}", lazy.IsValueCreated);

// The object can be used.

Console.WriteLine("Length = {0}", test.Length);

}

}

**Output**

IsValueCreated = False

Test()

IsValueCreated = True

Length = 10

**Internals.** The most interesting part of types in .NET is their internal implementations. This is an efficient way to learn about making types. For Lazy, the Value property has some elaborate logic that constructs the lazy object.  
**The Lazy type** accounts for boxed value types in many places. The methods LazyInitValue and then CreateValue typically lead to a call to Activator.CreateInstance, which essentially is a convoluted way to invoke the target constructor.  
**Performance.** There is substantial code complexity internal to the Lazy type and this will result in overhead and a performance penalty. Therefore, it is advisable that you only use Lazy on slower, hefty objects.  
**For example,** if an object must read a file, then Lazy will help performance because file IO is slow. Memory operations, though, would not benefit from Lazy initialization to the same extent.[**Memory Hierarchy**](https://www.dotnetperls.com/memory-hierarchy)  
**Constructors.** In an overloaded constructor, you can specify thread safety, and even specify a Func type that serves as a factory design pattern method. The Func will be invoked when the Value property is referenced.[**Func**](https://www.dotnetperls.com/func)  
**Concept.** Lazy evaluation is an established concept. Languages such as Lisp use lazy initialization. This is described in depth in the chapter on metalinguistic abstraction in the Structure and Interpretation of Computer Programs.

**Thunks:**The delayed objects are called thunks, and the term forcing refers to the creation of the lazy object (page 401).

**Summary.** Lazy programmer should consider the Lazy type. This type presents a reliable lazy initialization capability. It can reduce startup times in your C# programs in cases where its overhead is dwarfed by the cost of the initialization.

In c# a static class cannot implement an interface. When a single instance class needs to implement an interface for a business contracts or IoC purposes, this is where I use the Singleton pattern without a static class.

**Wrapper Pattern**

Hide the complexity of other classes, and protect yourself from future changes in the libraries your programs use, by using the wrapper/facade design pattern.

**Strategy Pattern**

When you need to perform a function several different ways, the Strategy Design Pattern might help you keep your code simpler and cleaner.

**Dependency Injection Pattern**

Use Dependency Injection to create automated tests for your business logic, without using any external resources (database, file system, etc.)

[Strategy pattern](http://sourcemaking.com/design_patterns/strategy) is one of the most useful design patterns in OOP. It lets you select an algoritm’s implementation at runtime

The Factory Pattern