[What is the difference between a mutable and immutable string in C#?](http://stackoverflow.com/questions/4274193/what-is-the-difference-between-a-mutable-and-immutable-string-in-c)

* a mutable string can be changed, and
* an immutable string cannot be changed.
* **String is immutable**
* i.e. strings cannot be altered. When you alter a string (by adding to it for example), you are actually creating a new string.
* **But StringBuilder is not immutable (rather, it is mutable)**
* so if you have to alter a string many times, such as multiple concatenations, then use StringBuilder.

# [How do I create an immutable Class?](http://stackoverflow.com/questions/352471/how-do-i-create-an-immutable-class)

* **all information injected into the class should be supplied in the constructor**
* **all properties should be getters only**
* if a collection (or Array) is passed into the constructor, it should be copied to keep the caller from modifying it later
* if you're going to return your collection, either return a copy or a read-only version (for example, using [ArrayList.ReadOnly](http://msdn.microsoft.com/en-us/library/system.collections.arraylist.readonly.aspx) or similar - you can combine this with the previous point and *store* a read-only copy to be returned when callers access it), return an enumerator, or use some other method/property that allows read-only access into the collection

|  |  |
| --- | --- |
| 11down vote | To be immutable, all your properties and fields should be readonly. And the items in any list should themselves be immutable.  You can make a readonly list property as follows:  public class MyClass  {  public MyClass(..., IList<MyType> items)  {  ...  \_myReadOnlyList = new List<MyType>(items).AsReadOnly();  }  public IList<MyType> MyReadOnlyList  {  get { return \_myReadOnlyList; }  }  private IList<MyType> \_myReadOnlyList  } |

# [SQL Server Race Condition](http://stackoverflow.com/questions/1683829/sql-server-race-condition-question)

Set the Transaction Isolation Level to Serializable.  
At lower isolation levels, other transactions can read the data in a row that is read, (but not yet modified) in this transaction. So two transactions can indeed read the same value. At very low isolation (Read Uncommitted) other transactions can even read data after it's been modified (but before committed)...

Review details about SQL Server Isolation Levels [here](http://msdn.microsoft.com/en-us/library/ms173763.aspx)

So bottom line is that the Isolation level is crtitical piece here to control what level of access other transactions get into this one

SET TRANSACTION ISOLATION LEVEL

    { READ UNCOMMITTED

    | READ COMMITTED

    | REPEATABLE READ

    | SNAPSHOT

    | SERIALIZABLE

    }

[ ; ]

READ UNCOMMITTED

Specifies that statements can read rows that have been modified by other transactions but not yet committed.

Transactions running at the READ UNCOMMITTED level do not issue shared locks to prevent other transactions from modifying data read by the current transaction. READ UNCOMMITTED transactions are also not blocked by exclusive locks that would prevent the current transaction from reading rows that have been modified but not committed by other transactions. When this option is set, it is possible to read uncommitted modifications, which are called dirty reads

READ COMMITTED

Specifies that statements cannot read data that has been modified but not committed by other transactions. This prevents dirty reads. Data can be changed by other transactions between individual statements within the current transaction, resulting in nonrepeatable reads or phantom data. **This option is the SQL Server default**.

REPEATABLE READ

Specifies that statements cannot read data that has been modified but not yet committed by other transactions and that no other transactions can modify data that has been read by the current transaction until the current transaction completes.

SNAPSHOT

Specifies that data read by any statement in a transaction will be the transactionally consistent version of the data that existed at the start of the transaction. The transaction can only recognize data modifications that were committed before the start of the transaction. Data modifications made by other transactions after the start of the current transaction are not visible to statements executing in the current transaction.

|  |
| --- |
| **System_CAPS_noteNote** |
| Under snapshot isolation mode, FILESTREAM data read by any statement in a transaction will be the transactionally consistent version of the data that existed at the start of the transaction, not at the start of the statement. |

SERIALIZABLE

Specifies the following:

* Statements cannot read data that has been modified but not yet committed by other transactions.
* No other transactions can modify data that has been read by the current transaction until the current transaction completes.
* Other transactions cannot insert new rows with key values that would fall in the range of keys read by any statements in the current transaction until the current transaction completes.
* [Examples](javascript:void(0))
* USE AdventureWorks2012;
* GO
* SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
* GO
* BEGIN TRANSACTION;
* GO
* SELECT \*
* FROM HumanResources.EmployeePayHistory;
* GO
* SELECT \*
* FROM HumanResources.Department;
* GO
* COMMIT TRANSACTION;
* GO

# [.NET HashTable Vs Dictionary - Can the Dictionary be as fast?](http://stackoverflow.com/questions/1089132/net-hashtable-vs-dictionary-can-the-dictionary-be-as-fast)

|  |  |
| --- | --- |
| 223down voteaccepted | System.Collections.Generic.Dictionary<TKey, TValue> and System.Collections.Hashtableclasses both maintain a hash table data structure internally. **None of them guarantee preserving the order of items.**  Leaving boxing/unboxing issues aside, most of the time, they should have very similar performance.  The primary structural difference between them is that Dictionary relies on *chaining* (maintaining a list of items for each hash table bucket) to resolve collisions whereas Hashtable uses *rehashing* for collision resolution (when a collision occurs, tries another hash function to map the key to a bucket).  There is little benefit to use Hashtable class if you are targeting for .NET Framework 2.0+. It's effectively rendered obsolete by Dictionary<TKey, TValue>. |

### C# Program to find whether the Number is Palindrome or not

num = Convert.ToInt32(Console.ReadLine());

temp = num;

**while** (Convert.ToBoolean(num))

{

rem = num % 10; //for getting remainder by dividing with 10

num = num / 10; //for getting quotient by dividing with 10

sum = sum \* 10 + rem; /\*multiplying the sum with 10 and adding

                           remainder\*/

}

# Lambda Expressions

 lambda expression is an [anonymous function](https://msdn.microsoft.com/en-IN/library/0yw3tz5k.aspx) that you can use to create [delegates](https://msdn.microsoft.com/en-IN/library/ms173172.aspx) or [expression tree](https://msdn.microsoft.com/en-IN/library/bb397951.aspx) types. By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls. Lambda expressions are particularly helpful for writing LINQ query expressions.

To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator [=>](https://msdn.microsoft.com/en-IN/library/bb311046.aspx), and you put the expression or statement block on the other side. For example, the lambda expression x => x \* x specifies a parameter that’s namedx and returns the value of x squared. You can assign this expression to a delegate type, as the following example shows

delegate int del(int i);

static void Main(string[] args)

{

del myDelegate = x => x \* x;

int j = myDelegate(5); //j = 25

}

# Asynchronous programming and Threading in C# (.NET 4.5)

<https://msdn.microsoft.com/en-us/library/hh191443.aspx>

<http://www.codeproject.com/Articles/996857/Asynchronous-programming-and-Threading-in-Csharp-N>

Asynchronous operation means that the operation runs independent of main or other process flow. In general c# program starts executing from the Main method and ends when the Main method returns. In between all the operations runs sequentially one after another. One operation must wait until its previous operation finishes. Let’s see following code:

Hide   Copy Code

static void Main(string[] args)

{

DoTaskOne();

DoTaskTwo();

}

Method “DoTaskTwo” would not be started until “DoTaskOne” finishes. In other words method “DoTaskOne” blocks the execution as long it takes to finish.

In asynchronous programming a method is called that runs in the background and the calling thread is not blocked. After calling the method the execution flow immediately backs to calling thread and performs other tasks. Normally it uses Thread or Task (We will discuss Thread and Task in detail later).

## Threading is required or not

If we use asynchronous programming pattern that .NET introduced in 4.5, in most of the cases we need not to create manual thread by us. The compiler does the difficult work that the developer used to do.

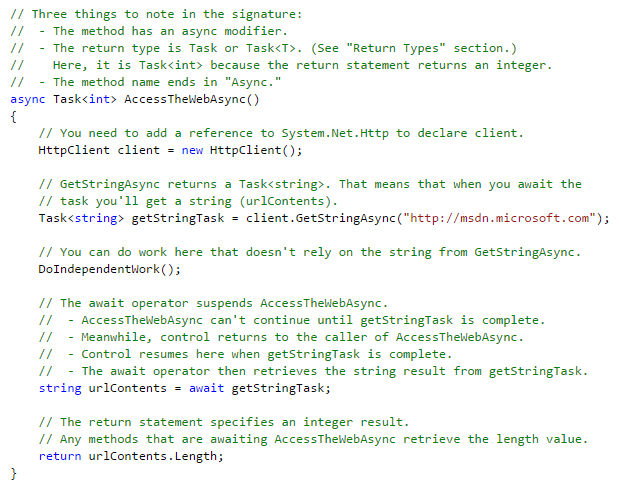
 Unless we need to control a thread, then “Task-based Asynchronous Pattern (TAP)”

But Task can be run:

1. In the current thread
2. In a new thread
3. In a thread from the thread pool
4. Or even without any thread
5. static void Main(string[] args)
6. {
7. Thread thread = new Thread(DoTask);
9. thread.Name = "My new thread";*// Asigning name to the thread*
10. thread.IsBackground = false;*// Made the thread forground*
11. thread.Priority = ThreadPriority.AboveNormal;*// Setting thread priority*
12. thread.Start();*// Start DoTask method in a new thread*
13. *//Do other task in main thread*
14. }

[Async Methods Are Easier to Write](javascript:void(0)" \o ")

The [Async](https://msdn.microsoft.com/en-us/library/hh191564.aspx) and [Await](https://msdn.microsoft.com/en-us/library/hh191564.aspx) keywords in Visual Basic and the [async](https://msdn.microsoft.com/en-us/library/hh156513.aspx) and [await](https://msdn.microsoft.com/en-us/library/hh156528.aspx) keywords in C# are the heart of async programming. By using those two keywords, you can use resources in the

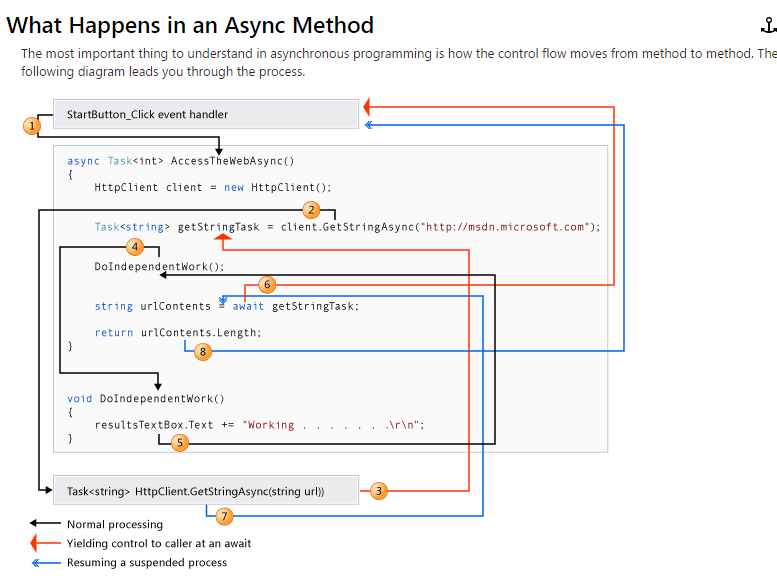


The following characteristics summarize what makes the previous example an async method.

* The method signature includes an **Async** or **async** modifier.
* The name of an async method, by convention, ends with an "Async" suffix.
* The return type is one of the following types:
  + [Task<TResult>](https://msdn.microsoft.com/en-us/library/dd321424.aspx) if your method has a return statement in which the operand has type TResult.
  + [Task](https://msdn.microsoft.com/en-us/library/system.threading.tasks.task.aspx) if your method has no return statement or has a return statement with no operand.
  + Void (a [Sub](https://msdn.microsoft.com/en-us/library/831f9wka.aspx) in Visual Basic) if you're writing an async event handler.

For more information, see "Return Types and Parameters" later in this topic.

* The method usually includes at least one await expression, which marks a point where the method can't continue until the awaited asynchronous operation is complete. In the meantime, the method is suspended, and control returns to the method's caller. The next section of this topic illustrates what happens at the suspension point.



The numbers in the diagram correspond to the following steps.

1. An event handler calls and awaits the AccessTheWebAsync async method.
2. AccessTheWebAsync creates an [HttpClient](https://msdn.microsoft.com/en-us/library/system.net.http.httpclient.aspx) instance and calls the [GetStringAsync](https://msdn.microsoft.com/en-us/library/hh551746.aspx) asynchronous method to download the contents of a website as a string.
3. Something happens in **GetStringAsync** that suspends its progress. Perhaps it must wait for a website to download or some other blocking activity. To avoid blocking resources, **GetStringAsync** yields control to its caller, AccessTheWebAsync.

**GetStringAsync** returns a [Task<TResult>](https://msdn.microsoft.com/en-us/library/dd321424.aspx) where TResult is a string, and AccessTheWebAsync assigns the task to thegetStringTask variable. The task represents the ongoing process for the call to **GetStringAsync**, with a commitment to produce an actual string value when the work is complete.

1. Because getStringTask hasn't been awaited yet, AccessTheWebAsync can continue with other work that doesn't depend on the final result from **GetStringAsync**. That work is represented by a call to the synchronous method DoIndependentWork.
2. DoIndependentWork is a synchronous method that does its work and returns to its caller.
3. AccessTheWebAsync has run out of work that it can do without a result from getStringTask. AccessTheWebAsync next wants to calculate and return the length of the downloaded string, but the method can't calculate that value until the method has the string.

Therefore, AccessTheWebAsync uses an await operator to suspend its progress and to yield control to the method that called AccessTheWebAsync. AccessTheWebAsync returns a **Task(Of Integer)** or **Task<int>** to the caller. The task represents a promise to produce an integer result that's the length of the downloaded string.

|  |
| --- |
| **System_CAPS_noteNote** |
| If **GetStringAsync** (and therefore getStringTask) is complete before AccessTheWebAsync awaits it, control remains inAccessTheWebAsync. The expense of suspending and then returning to AccessTheWebAsync would be wasted if the called asynchronous process (getStringTask) has already completed and AccessTheWebSync doesn't have to wait for the final result. |

Inside the caller (the event handler in this example), the processing pattern continues. The caller might do other work that doesn't depend on the result from AccessTheWebAsync before awaiting that result, or the caller might await immediately. The event handler is waiting for AccessTheWebAsync, and AccessTheWebAsync is waiting for **GetStringAsync**.

1. **GetStringAsync** completes and produces a string result. The string result isn't returned by the call to **GetStringAsync** in the way that you might expect. (Remember that the method already returned a task in step 3.) Instead, the string result is stored in the task that represents the completion of the method, getStringTask. The await operator retrieves the result fromgetStringTask. The assignment statement assigns the retrieved result to urlContents.
2. When AccessTheWebAsync has the string result, the method can calculate the length of the string. Then the work ofAccessTheWebAsync is also complete, and the waiting event handler can resume. In the full example at the end of the topic, you can confirm that the event handler retrieves and prints the value of the length result.

If you are new to asynchronous programming, take a minute to consider the difference between synchronous and asynchronous behavior. A synchronous method returns when its work is complete (step 5), but an async method returns a task value when its work is suspended (steps 3 and 6). When the async method eventually completes its work, the task is marked as completed and the result, if any, is stored in the task.

For more information about control flow, see [Control Flow in Async Programs (C# and Visual Basic)](https://msdn.microsoft.com/en-us/library/hh873191.aspx).

**THREADS**

Async methods are intended to be non-blocking operations. An await expression in an async method doesn’t block the current thread while the awaited task is running. Instead, the expression signs up the rest of the method as a continuation and returns control to the caller of the async method.

The async and await keywords don't cause additional threads to be created. Async methods don't require multithreading because an async method doesn't run on its own thread. The method runs on the current synchronization context and uses time on the thread only when the method is active. You can use [Task.Run](https://msdn.microsoft.com/en-us/library/hh195051.aspx) to move CPU-bound work to a background thread, but a background thread doesn't help with a process that's just waiting for results to become available.

* The marked async method can use [Await](https://msdn.microsoft.com/en-us/library/hh156570.aspx) or [await](https://msdn.microsoft.com/en-us/library/hh156528.aspx) to designate suspension points. The await operator tells the compiler that the async method can't continue past that point until the awaited asynchronous process is complete. In the meantime, control returns to the caller of the async method.

The suspension of an async method at an await expression doesn't constitute an exit from the method, and **finally** blocks don’t run.

* The marked async method can itself be awaited by methods that call it.

An async method typically contains one or more occurrences of an await operator, but the absence of await expressions doesn’t cause a compiler error. If an async method doesn’t use an await operator to mark a suspension point, the method executes as a synchronous method does, despite the async modifier. The compiler issues a warning for such methods.