………………………………………………………………………………….............................................................................

…………………………………………….**Func, Action, Predicate, Delegate, Lambda**…………………………………………

**Delegate**—pass method as a parameter with some defined syntax and using delegate keyword

**Anonymous function**-- we can pass method body only just using delegate keyword with any delegate declaration

**Lambda functions**—removed delegate keyword from anonymous function and make this easy to use

**Multicast delegate v/s** reflection : Delegate call invoke methods just by passing name of methods and parameter while Reflection does the same but it doesn’t need methods parameter so if parameter changes but still we don’t need to make any change in methods call as it takes only methods name.

C# - Delegate

A function can have one or more parameters of different data types, but what if you want to pass a function itself as a parameter? How does C# handle the callback functions or event handler? The answer is - delegate.

……………………………………………………………………………………

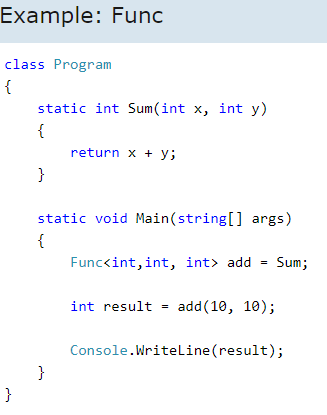
https://www.tutorialsteacher.com/Content/images/bulb-glow.png Points to Remember:

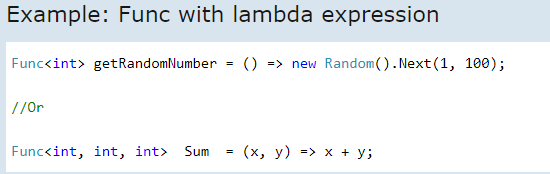
1. Delegate is a function pointer. It is reference type data type.
2. Syntax: public delegate void <function name>(<parameters>)
3. A method that is going to assign to delegate must have same signature as delegate.
4. Delegates can be invoke like a normal function or Invoke () method.
5. Multiple methods can be assigned to the delegate using "+" operator. It is called multicast delegate.

# C# - Func

C# 3.0 includes built-in generic delegate types Func and Action, so that you don't need to define custom delegates as above.

Func is a generic delegate included in the System namespace. It has zero or more input parameters and one out parameter. The last parameter is considered as an out parameter.



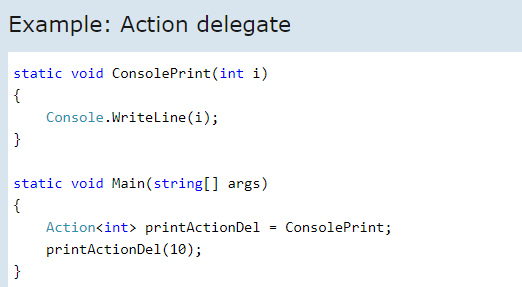


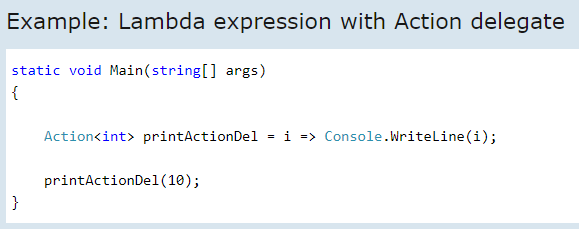
https://www.tutorialsteacher.com/Content/images/bulb-glow.png Points to Remember:

1. Func is built-in delegate type.
2. Func delegate type must return a value.
3. Func delegate type can have zero to 16 input parameters.
4. Func delegate does not allow ref and out parameters.
5. Func delegate type can be used with an [anonymous method](https://www.tutorialsteacher.com/csharp/csharp-anonymous-method) or [lambda expression](https://www.tutorialsteacher.com/linq/linq-lambda-expression).

# C# - Action Delegate

Action is also a delegate type defined in the System namespace. An Action type delegate is the same as [Func delegate](https://www.tutorialsteacher.com/csharp/csharp-func-delegate) except that the Action delegate doesn't return a value. In other words, an Action delegate can be used with a method that has a void return type.





## Advantages of Action and Func Delegates

1. Easy and quick to define delegates.
2. Makes code short.
3. Compatible type throughout the application.

https://www.tutorialsteacher.com/Content/images/bulb-glow.png Points to Remember :

1. Action delegate is same as func delegate except that it does not return anything. Return type must be void.
2. Action delegate can have 1 to 16 input parameters.
3. Action delegate can be used with [anonymous methods](https://www.tutorialsteacher.com/csharp/csharp-anonymous-method) or [lambda expressions](https://www.tutorialsteacher.com/linq/linq-lambda-expression).

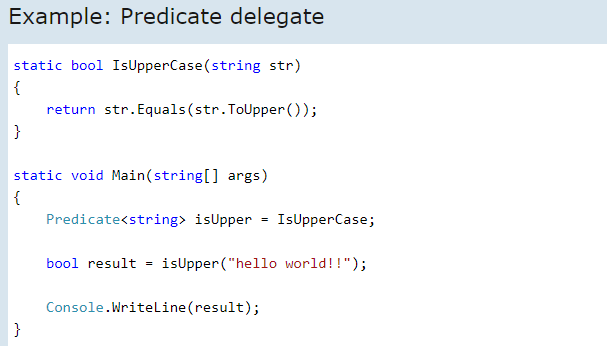
# C# - Predicate Delegate

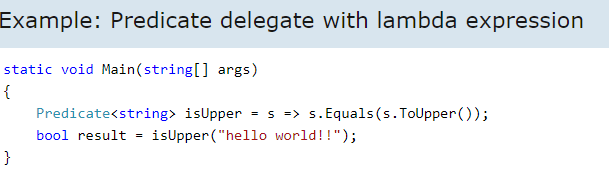
A predicate is also a delegate like [Func](https://www.tutorialsteacher.com/csharp/csharp-func-delegate) and [Action](https://www.tutorialsteacher.com/csharp/csharp-action-delegate) delegates. It represents a method that contains a set of criteria and checks whether the passed parameter meets those criteria or not. Predicate delegate methods must take one input parameter and return a Boolean - true or false.

The Predicate delegate is defined in the System namespace as shown below:

Predicate signature: public delegate bool Predicate<in T>(T obj);

Same as other delegate types, Predicate can also be used with any method, anonymous method or lambda expression.





https://www.tutorialsteacher.com/Content/images/bulb-glow.png Points to Remember:

1. Predicate delegate takes one input parameter and Boolean return type.
2. [Anonymous method](https://www.tutorialsteacher.com/csharp/csharp-anonymous-method) and [Lambda expression](https://www.tutorialsteacher.com/linq/linq-lambda-expression) can be assigned to the predicate delegate.

**ExecuteScalar()** only returns the value from the first column of the first row of your query.  
**ExecuteReader()** returns an object that can iterate over the entire result set while only keeping one record in memory at a time.  
**ExecuteNonQuery**() does not return data at all: only the number of rows affected by an insert, update, or delete.

Additionally, you can look at the DbDataAdapter type, which includes a Fill() method that allows you to download the entire resultset into a DataTable or DataSet object, as well as a number of other abilities.

It depends on your needs. One of the most important differences is that a DataReader will retain an open connection to your database until you’re done with it while a DataSet will be an in-memory object. If you bind a control to a DataReader then it’s still open. In addition, a DataReader is a forward only approach to reading data that can’t be manipulated. With a DataSet you can move back and forth and manipulate the data as you see fit.

Some additional features: DataSets can be serialized and represented in XML and, therefore, easily passed around to other tiers. DataReaders can’t be serialized.

On the other hand if you have a large amount of rows to read from the database that you hand off to some process for a business rule a DataReader may make more sense rather than loading a DataSet with all the rows, taking up memory and possibly affecting scalability.

The DataSet class in ADO.Net operates in an entirely disconnected nature, while DataReader is a connection oriented service.

DataSet is an in-memory representation of a collection of Database objects including related tables, constraints, and relationships among the tables. It provides a consistent relational programming model with multiple data sources from different areas. We can say that the DataSet is a small database because it stores the schema and data in the application memory area. Dataset is used to hold tables with data. You can select data form tables, create views based on table and ask child rows over relations. Also DataSet provides you with rich features like saving data as XML and loading XML data.

DataReader is designed to retrieve a read-only, forward-only stream of data from data sources. DataReader has a connection oriented nature, whenever you want fetch the data from database that you must have a connection. It’s usually the most efficient way to deal with records when you don’t need any random access. It fetches one row at a time so very less network cost when compare to DataSet. Results are returned as the query executes, and are stored in the network buffer on the client until you request them using the Read method of the DataReader. DataReader is readonly so we can’t do any transaction on them. It will be the best choice where we need to show the data to the user which requires no transaction.

The biggest drawbacks of DataSet is speed because it is a high resource consuming process. It carrying considerable overhead because of related tables, constraints, and relationships among the tables. If you need forward only access to query results then DataReader is the best choice because in this scenario it is fastest way to go.

………………………………………………………………………………………………………………………………………………………….

As David implies, there difference between String.Empty and “” are pretty small, but there is a difference. “” actually creates an object, it will likely be pulled out of the string intern pool, but still… while String.Empty creates no object… so if you are really looking for ultimately in memory efficiency, I suggest String.Empty. However, you should keep in mind the difference is so trival you will like never see it in your code…

---------------

Both int.Parse and Convert.ToInt32 are used to convert string into the integer but Only difference between them is to Convert.ToInt32 handle null and returns '0' as output and int.parse is not going to handle NULL and will give a Argument Null Exception. Here is the example for that both are almost same except handling null.

-----------

**Decimal v/s Double v/s Float:**

Precision is the main difference

Float - 32 bit (7 digits)

Double - 64 bit (15-16 digits)

Decimal - 128 bit (28-29 significant digits)

**Decimal**: In case of financial applications it is better to use Decimal types because it gives you a high level of accuracy and easy to avoid rounding errors

**Double:** Double Types are probably the most normally used data type for real values, except handling money.

**Float**: It is used mostly in graphic libraries because very high demands for processing powers also used situations that can endure rounding errors.

**When static method:**

Generally static variables declared for which field is common or share to the all instance of the class but i want when to use static methods c#

You use static methods when the method does not need to access any non-static class elements such as properties, events, or methods in order to do its job.

Think of it this way: how many wheels does a car have? What colour is a car?

The first is a static question, because all cars have 4 wheels (if they had two they would be motorcycles, and three would make them tricycles). You don't need to establish which car you are talking about, because the question is generic to all cars.

The second is non-static: you have to indicate which car you are talking about. "My Car" is black; "Your Car" is green; "This car" is red; "That car" is blue - you cannot answer the question unless you know which car you are talking about.

Classes are the same: static methods (and properties, fields, events) when you don't need information about a specific instance, non-static when you do.

You should always use static when you don't need an object around you method, and use dynamic when you need an object. In the example you provides, you don't need an object, because the method doesn't interact with any properties or fields in your class.

Making a class static just prevents people from trying to make an instance of it. If all your class has are static members it is a good practice to make the class itself static.

------------

# Introduction

To compare equality between variables C# has provided two ways of doing comparison “==” and an overloaded method “equals()”. Most of the developers use “==” and “Equals” is hardly discussed.

# Point 1 :- Comparison on the basis of Equality:

Answering to the point “There is no difference between equality comparison using “==” and “Equals()”, **except when you are comparing “String” comparison.**

The common comparison Rule:-Whenever you are comparing variables they are either value types or reference types. When values types are compared they are compared on the basis of “Content” when reference types are compared they are compared on the basis of “Reference”(memory location) and not “Content”.

The above rule is respected by both “==” and “Equals”.

## Scenario 1:- Value type comparison

When you compare value types / primitive data types (int , double etc) either by using “==” or “Equals” it’s always based on content. In the below code you can see both comparison methods will show as “true”.

Hide   Copy Code

int i = 10;

int y = 10;

Console.WriteLine(i == y); *// true*

Console.WriteLine(i.Equals(y)); *// true*

## 

## Scenario 2:- Reference types comparison

Now when you compare objects they are compared on the basis of reference (internal memory pointer). Below obj and obj1 comparison either through “==” or “Equals” will be false. So in the below code even though both the object have property name as “Shiv” still it shows unequal. Because the comparison is based on internal memory reference which is different for “obj” and “obj1”

Hide   Copy Code

Customerobj = newCustomer();

obj.Name = "Shiv";

Customer obj1 = newCustomer();

obj1.Name = "Shiv";

Console.WriteLine(obj == obj1); *// false*

Console.WriteLine(obj.Equals(obj1)); *// false*

But the below code will display true as the pointer points to same object.

Hide   Copy Code

Customerobj = newCustomer();

obj.Name = "Shiv";

Customer obj1 = obj;

Console.WriteLine(obj == obj1); *// true*

Console.WriteLine(obj.Equals(obj1)); *// true*

## 

## Scenario 3:- String comparison, interning and object type casting

Now strings are immutable objects or reference types so they should be checked using the rules of reference types. In other words in the below scenario when we assign value to “str” it creates a string object and in heap has “test” stored. When you now assign “str1” this a different objects so it should be a different instance.

But look at the value, it the same. So C# string follows interning rule. In other words if the content is same “str” and “str1” they point to the same memory location and data. So both “==” and “Equals” will be true.

Hide   Copy Code

objectstr = "test";

object str1 = "test";

Console.WriteLine(str==str1);

Console.WriteLine(str.Equals(str1));

But now look at the below code where we are explicitly creating new separate objects of string with same value. We are forcing and overriding interning behavior of string.In the below code “==” will return false even though the content is same while “Equals” will return true. This is one place where the equality behavior differs.

Hide   Copy Code

objectstr = newstring(newchar[] { 't', 'e', 's', 't' });

object str1 = newstring(newchar[] { 't', 'e', 's', 't' });

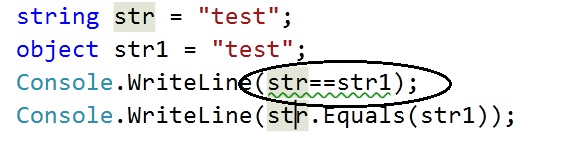
Console.WriteLine(str==str1); *// false*

Console.WriteLine(str.Equals(str1)); *// true*

# 

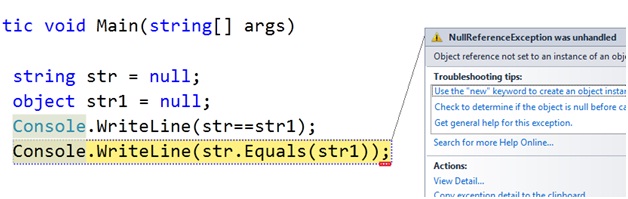
# Point 2 :- Compile time VS RunTime

The next point which makes them different is when do type checks happen. “==” does type checking during compile time while “Equals” is more during runtime. You can see in the below code how “==” is showing a warning message with green sign saying that you are comparing different types and you can have issues. “Equals” does not show any such warnings.



# Point 3 :- The NULL Situation

“==” works with nulls but “Equals” crashes when you compare NULL values, see the below print screen.



# The Full Picture

So if we list down all the point the final conclusion is the below table.

|  | **==** | **Equals** |
| --- | --- | --- |
| **Usage** | Technical based | Semantic based |
| **Value types** | Content based Comparison | Content based Comparison |
| **Objects** | Reference based Comparison | Reference based Comparison |
| **String** | Content based Comparison | Content based Comparison |
| **String with no interning** | Reference based Comparison | Content based Comparison |
| **Type checking** | Compile time | Run time |
| **Nulls** | Works | Can crash |

**Immutable v/c Mutable**

Mutable and immutable are English words that mean "can change" and "cannot change" respectively. The meaning of these words is the same in C# programming language; that means the mutable types are those whose data members can be changed after the instance is created but Immutable types are those whose data members can’t be changed after the instance is created.

When we change the value of mutable objects, value is changed in same memory. But in immutable type, the new memory is created and the modified value is stored in new memory.

Strings are immutable, which means we are creating new memory everytime instead of working on existing memory.

We have some advantages of immutable classes like immutable objects are simpler to construct, test, and use. Immutable objects are always thread-safe and etc.

StringBuilder is a mutable type that means we are using the same memory location and keep on appending/modifying the stuff to one instance. It will not create any further instances hence it will not decrease the performance of the application.

**Immutable Class: (We can’t change anything from outside)**

1. **class** MyClass
2. {
3. **private** readonly string myStr;
5. **public** MyClass(string str)
6. {
7. myStr = str;
8. }
10. **public** string GetStr
11. {
12. get { **return** myStr; }
13. }
14. }