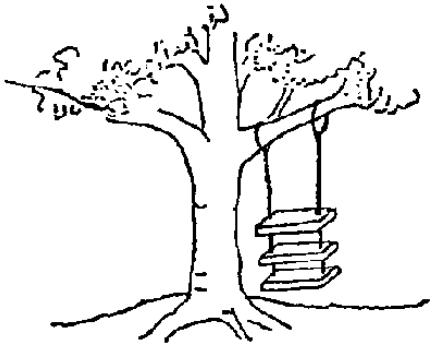
Delegate & Event Lambda Expression

Software Entwicklung



Overview

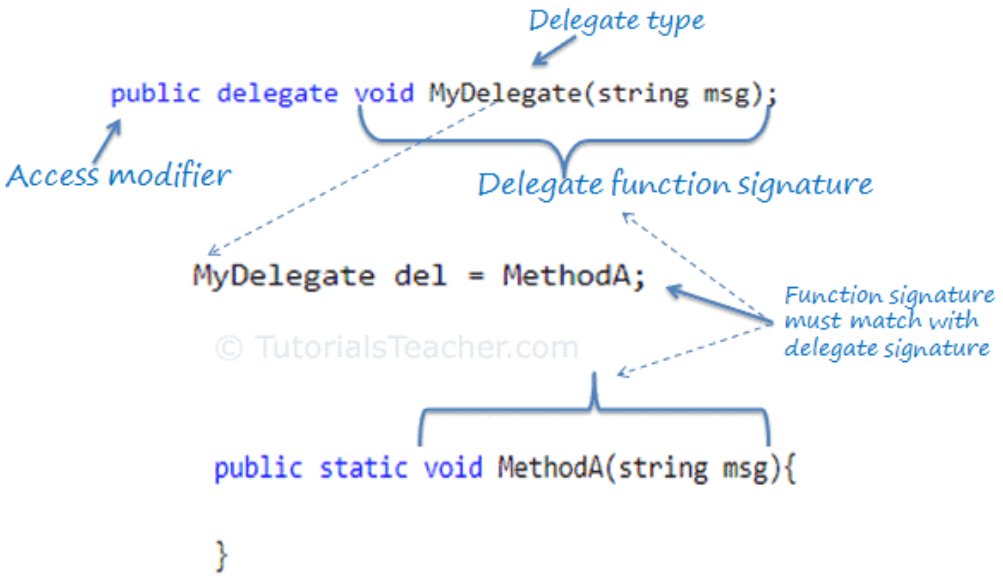
- Delegate
- Delegate as Parameter
- Multicast Delegate
- Events

- Action Funk Predicate

- Lambda Expression

chosen to represent others

chosen to represent others



Delegate

is a reference type data type that defines the method signature

define variables of delegate, just like other data type, that can refer to any method with the same signature as the delegate.

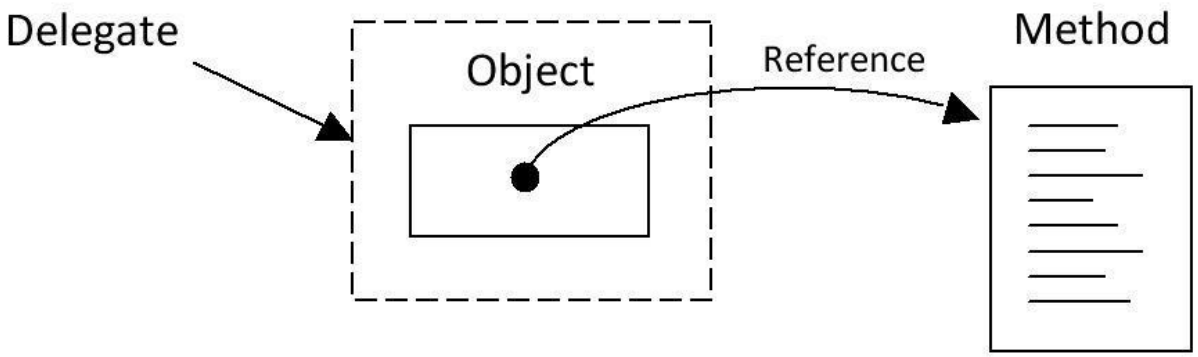
http://www.tutorialsteacher.com/csharp/csharp-delegates

Motivation of Delegates

- 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**.

Definition of Delegate

- is a reference type variable
  - holds the **reference to a method**
  - reference can be changed at runtime



- used for implementing events and the **call-back methods**
- are implicitly derived from **System.Delegate**

Declaring Delegates

- Delegate can refer to a method, which has the same signature as the delegate

- **Example:**

```
public delegate int MyDelegate (string s);
```

- **Syntax:**

```
access_modifier delegate <return type>
                <delegate-name> (<parameter list>)
```

Instantiating Delegates

- created with new keyword
- associated with a particular method
- the argument passed to the new expression is written similar to a method call, but without the arguments to the method

- **Example:**

```
public delegate void printString(string s);
...
printString ps1 = new printString(WriteToScreen);
printString ps2 = WriteToFile;
```

Invoking Delegate

Delegates

- can be invoke like a normal function
- or Invoke() method

- **Example:**

```
Print printDel = PrintNumber;
//using () operator
printDel (200);

//using the Invoke() method of delegate
printDel.Invoke(200);
```

```
Number: 200
Number: 200
```

Working with delegates

```
static void MethodA(string message) {
 Console.WriteLine(message);
}
```

1. Declare a delegate

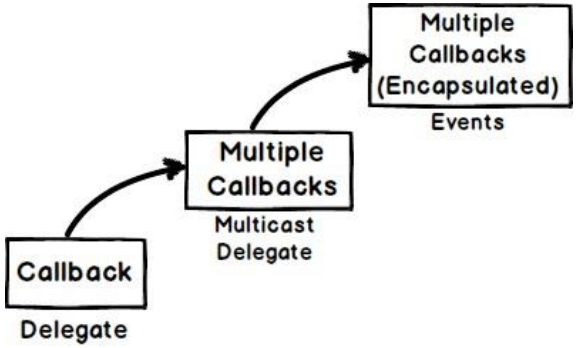
```
public delegate void MyDelegate(string msg);
```

2. Set a target method

```
MyDelegate del = new MyDelegate(MethodA);
// or
MyDelegate del = MethodA;
```

3. Invoke a delegate

```
del.Invoke("Hello World!");
// or
del("Hello World!");
```



First Example with Delegate

- Static Method – same Class

```
class Program {
    public delegate void MyDelegate(string msg); // declare a delegate
    static void Main(string[] args) {
        // set target method
        MyDelegate del = new MyDelegate(MethodA);
        // or
        MyDelegate dell = MethodA;

        del.Invoke("Hello World!");
        // or
        dell("Hello World!");
    }
    // target method
    static void MethodA(string message) {
        Console.WriteLine(message);
    }
}
```

```
Hello World!
Hello World!
```

First Example with Lambda

```
// declare a delegate
public delegate void MyDelegate(string msg);

static void Main(string[] args) {
    // set target method
    MyDelegate del1 = new MyDelegate(MethodA);
    // or
    MyDelegate del2 = MethodA;
    // or set lambda expression
    MyDelegate del3 = (string msg) => Console.WriteLine(msg);

    del1.Invoke("Hello World!");
    // or
    del2("Hello World!");
    del3("Hallo Welt!");
}
// target method
static void MethodA(string message) {
    Console.WriteLine(message);
}
```

Second Example

```
public delegate void MyDelegate(string msg); //declaring a delegate

class ProgrammDel2 {
    static void Main2(string[] args) {
        MyDelegate del = ClassA.MethodA;
        del("Hello World");

        del = new ClassB().MethodB;
        del("Hello World");

        del = (string msg) => Console.WriteLine("Called lambda expression: " + msg);
        del("Hello World");
    }
}
class ClassA {
    public static void MethodA(string message) {
        Console.WriteLine("Called ClassA.MethodA() with parameter: " + message);
    }
}
class ClassB {
    public void MethodB(string message) {
        Console.WriteLine("Called MethodB() from ClassB with parameter: " + message);
    }
}
```

static method
or dynamic method
in another class

Passing Delegate as a Parameter

```
class Program {
    static void Main(string[] args) {
        MyDelegate del = ClassA.MethodA;
        InvokeDelegate(del);

        del = new ClassB().MethodB;
        InvokeDelegate(del);

        del = (string msg) => Console.WriteLine("Called lambda expression: " + msg);
        InvokeDelegate(del);
    }

    static void InvokeDelegate(MyDelegate del) // MyDelegate type parameter
    {
        del("Hello World");
    }
}
```

```
class ClassA {
    public static void MethodA(string message) {
        Console.WriteLine("Called ClassA.MethodA() with parameter: " + message);
    }
}
class ClassB {
    public void MethodB(string message) {
        Console.WriteLine("Called MethodB() from ClassB with parameter: " + message);
    }
}
```

```
Called ClassA.MethodA() with parameter: Hello World
Called MethodB() from ClassB with parameter: Hello World
Called lambda expression: Hello World
```

Multicast Delegate

```
class ClassA {
    public static void MethodA(string message) {
        Console.WriteLine("Called ClassA.MethodA() with parameter: " + message);
    }
}
class ClassB {
    public void MethodB(string message) {
        Console.WriteLine("Called MethodB() from ClassB with parameter: " + message);
    }
}
public delegate void MyDelegate(string msg); //declaring a delegate

class Program {
    static void Main(string[] args) {
        MyDelegate del1 = ClassA.MethodA;
        MyDelegate del2 = new ClassB().MethodB;

        MyDelegate del = del1 + del2; // combines del1 + del2
        del("Hello World");

        MyDelegate del3 = (string msg) => Console.WriteLine("Called lambda expression: " + msg);
        del += del3; // combines del1 + del2 + del3
        del("Hello World");

        del = del - del2; // removes del2
        del("Hello World");

        del -= del1;// removes del1
        del("Hello World");
    }
}
```

```
Called ClassA.MethodA() with parameter: Hello World
Called MethodB() from ClassB with parameter: Hello World
Called ClassA.MethodA() with parameter: Hello World
Called MethodB() from ClassB with parameter: Hello World
Called lambda expression: Hello World
Called ClassA.MethodA() with parameter: Hello World
Called lambda expression: Hello World
Called lambda expression: Hello World
```

Multicast Delegate with Parameter

- addition and subtraction operators always work as part of the assignment:
  - del1 += del2;
  - is exactly equivalent to
  - del1 = del1+del2;
  - and likewise for subtraction
- if a delegate returns a value:
  - **the last** assigned target method's value will be return

```
public delegate int MyDelegate(); //declaring a delegate

class Program {
    static void Main(string[] args) {
        MyDelegate del1 = ClassA.MethodA;
        MyDelegate del2 = ClassB.MethodB;

        MyDelegate del = del1 + del2;
        Console.WriteLine(del());// returns 200
    }
}

class ClassA {
    public static int MethodA() {
        return 100;
    }
}

class ClassB {
    public static int MethodB() {
        return 200;
    }
}
```

Generic Delegate

- generic delegate using generic type parameters or return type:

```
public delegate T add<T>(T param1, T param2); // generic delegate

class Program {
    static void Main(string[] args) {
        add<int> sum = Sum;
        Console.WriteLine(sum(10, 20));

        add<string> con = Concat;
        Console.WriteLine(con("Hello ", "World!!"));
    }

    public static int Sum(int val1, int val2) {
        return val1 + val2;
    }

    public static string Concat(string str1, string str2) {
        return str1 + str2;
    }
}
```

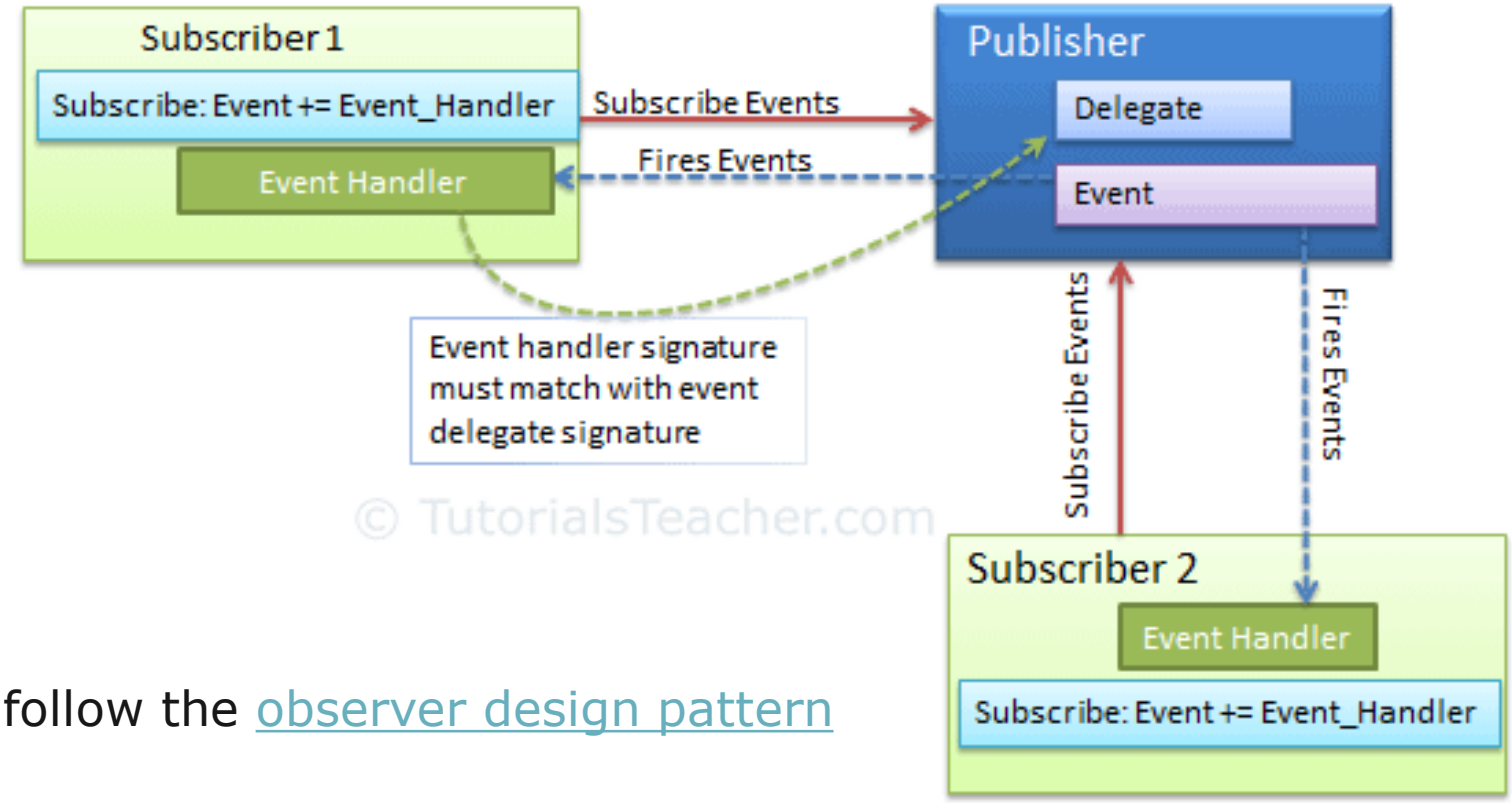
```
30
Hello World!!
```

Event

- is a notification sent by an object to signal the occurrence of an action
- is an encapsulated delegate
- is dependent on the delegate, which defines the signature for the event handler method of the subscriber class

Event

- figure illustrates the event in C#



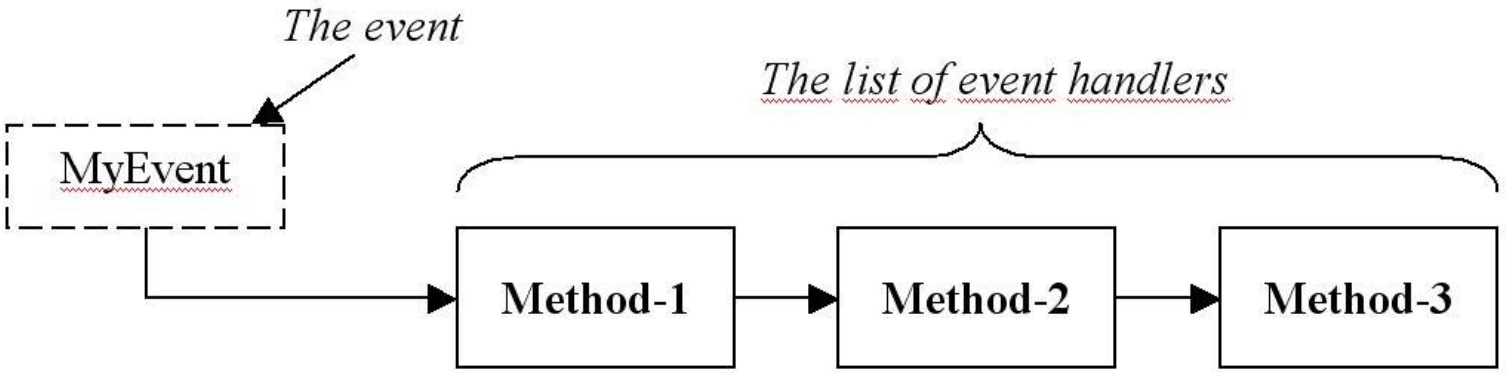
follow the observer design pattern

Observer Pattern
with Publisher & Subscriber

- something special that is going to happen:

  Publisher:
  - someone who launches (**raises**) an **event**
  - and **notifies** the others about it

  Subscriber:
  - the others are the **subscribers** of the event
  - and attend (**handle**) the event



Declare an Event:

- An event can be declared in two steps:
  - Declare a delegate.
  - Declare a variable of the delegate with event keyword.

- How to declare an event in publisher class…

```
public delegate void Notify();  // delegate

public class ProcessBusinessLogic
{
    public event Notify ProcessCompleted; // event

}
```

Raising an Event

```
public delegate void Notify();  // delegate

public class ProcessBusinessLogic
{
    public event Notify ProcessCompleted; // event

    public void StartProcess()
    {
        Console.WriteLine("Process Started!");
        // some code here..
        OnProcessCompleted();
    }

    protected virtual void OnProcessCompleted() //protected virtual method
    {
        //if ProcessCompleted is not null then call delegate
        ProcessCompleted?.Invoke();
    }
}
```

Points to remember

1. Delegate is the reference type data type that defines the signature.
2. Delegate type variable can refer to any method with the same signature as the delegate.
3. **Syntax:**
   [access modifier] delegate [return type] [delegate name]([parameters])
4. A target method's signature must match with delegate signature.
5. Delegates can be invoke like a normal function or Invoke() method.
6. Multiple methods can be assigned to the delegate using "+" or "+=" operator and removed using "-" or "-=" operator. It is called multicast delegate.
7. If a multicast delegate returns a value then it returns the value from the last assigned target method.
8. Delegate is used to declare an event and anonymous methods in C#.

delegate

lamda =>

Func<>

Action<>

predicate<>

Generic Function

Func Delegate

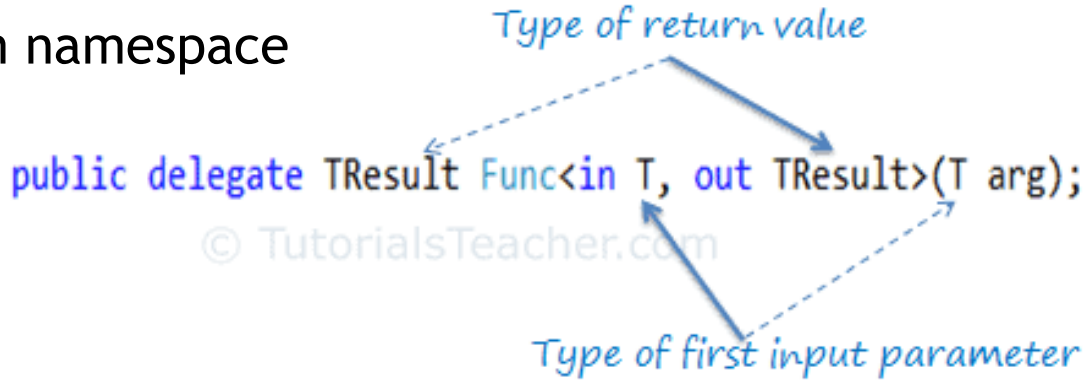
Action Delegate

Predicate Delegate

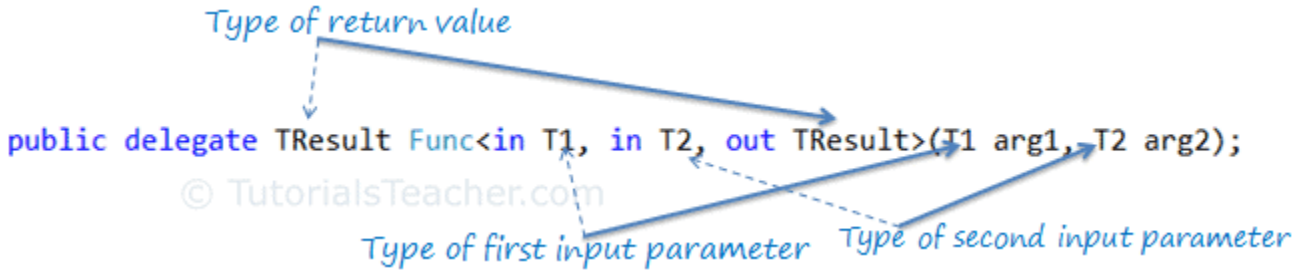
Anonymous Methods

Func - a generic delegate

- included in the System namespace



- has zero or more *input* parameters & one *out* parameter
- the last parameter is considered as an out parameter



http://www.tutorialsteacher.com/csharp/csharp-func-delegate

Generic Function Type: Func

- Func is the basic function object
  - Can be declared with a delegate or lambda

```
delegate double MyFunction(double x);
MyFunction f = Math.Sin;
double y = f(4); //y=sin(4)
```

- `Func<T1,T2,T3,...Tn> [name] = …`

```
Func<double, double> myFancyFunc = Math.Sin;
double y = myFancyFunc(4);
```

- The last parameter is the return type

Sum with Funk

```
Func<int, int, int> sum;

static int Sum(int x, int y) {
    return x + y;
}

static void Main(string[] args) {
    Func<int, int, int> add = Sum;

    int result = add(10, 10);

    Console.WriteLine(result);
}
```

Func delegate type can include 0 to 16 input parameters of different types.

```
//Sum with Lambda Expression
Func<int, int, int> Sum = (x, y) => x + y;
```

Random Number with Funk

```
static void Main() {
    Func<int> getRandomNumber = delegate () {
        Random rnd = new Random();
        return rnd.Next(1, 100);
    };


    int number = getRandomNumber();
    Console.WriteLine(number);


    //getRandomNumber with Lambda Expression
    Func<int> getRandomNumberL = () => new Random().Next(1, 100);
}
```

Funk delegate type must include an out parameter for the result.

```
public delegate void Action<in T>(
        T obj
)
```

Action

is a delegate type in the System namespace

- is the same as Func delegate except that it doesn‘t return a value
- can be used with a method that returns void

http://www.tutorialsteacher.com/csharp/csharp-action-delegate

Print with Func<int>

```
Action<int> printActionDel = new Action<int>(ConsolePrint);
```

```
public delegate void Print(int val);

static void ConsolePrint(int i)
{
    Console.WriteLine(i);
}

static void Main(string[] args)
{
    Print prnt = ConsolePrint;
    prnt(10);
}
```

How to use an Action<int> instead of a Print delegate…

```
static void ConsolePrint(int i)
{
    Console.WriteLine(i);
}

static void Main(string[] args)
{
    Action<int> printActionDel = ConsolePrint;
    printActionDel(10);
}
```

Using Func …

- With Anonymous Method:

```
public static void Main()
{
    Action<int> printActionDel = delegate(int i){ Console.WriteLine(i); };
    printActionDel(10);
}
```

- With Lambda Expression

```
public static void Main()
{
    Action<int> printActionDel = i => Console.WriteLine(i);
    printActionDel(10);
}
```

```
public delegate bool Predicate<in T>(
        T obj
)
```

Predicate

Predicate is the delegate like Func and Action delegates

represents a method containing a set of criteria and checks whether the passed parameter meets those criteria

must take one input parameter and return a boolean - true or false

delegate is defined in the System namespace

can also be used with any method, anonymous method, or lambda expression

http://www.tutorialsteacher.com/csharp/csharp-predicate

IsUpper with Predicate<string>

```
static bool IsUpperCase(string str)
{
    return str.Equals(str.ToUpper());
}

static void Main(string[] args)
{
    Predicate<string> isUpper = IsUpperCase;

    bool result = isUpper("hello world!!");

    Console.WriteLine(result);
}
```

Predicate delegate …

- with anonymous method

```
static void Main(string[] args)
{
    Predicate<string> isUpper = delegate(string s) { return s.Equals(s.ToUpper());};
    bool result = isUpper("hello world!!");
}
```

- with lambda expression

```
static void Main(string[] args)
{
    Predicate<string> isUpper = s => s.Equals(s.ToUpper());
    bool result = isUpper("hello world!!");
}
```

Func – Action - Predicate

- There are three types of functions:

  Func<> returns a value

  Action<> no return value (void)

  Predicate<> returns a bool

- Advantages of Action, Predicate & Func Delegates
  1. Easy and quick to define delegates.
  2. Makes code short.
  3. Compatible type throughout the application.

Points to Remember :

- **Func** delegate type must return a value.
- **Func** delegate type can have zero to 16 input parameters.
- **Func** delegate does not allow ref and out parameters.

- **Action** delegate is same as func delegate except that it does not return anything. Return type must be void.
- **Action** delegate can have 0 to 16 input parameters.

- **Predicate** delegate takes one input parameter and boolean return type.

- **Func, Action & Predicate**
  - are built-in delegate types
  - can be used with an anonymous method or lambda expression

Anonymous Method

is a method without a name

- can be defined using the delegate keyword
- can be assigned to a variable of delegate type

Anonymous Method

- can be defined using the delegate keyword and
- can be assigned to a variable of delegate type

```
public delegate void Print(int value);

static void Main(string[] args)
{
    Print print = delegate(int val) {
        Console.WriteLine("Inside Anonymous method. Value: {0}", val);
    };

    print(100);
}
```

```
Inside Anonymous method. Value: 100
```

Anonymous Method

- can access variables defined in an outer function

```
public delegate void Print(int value);

static void Main(string[] args)
{
    int i = 10;

    Print prnt = delegate(int val) {
        val += i;
        Console.WriteLine("Anonymous method: {0}", val);
    };

    prnt(100);
}
```

```
Anonymous method: 110
```

Anonymous Method as Parameter

- can also be passed to a method that accepts the delegate as a parameter

```
public delegate void Print(int value);

public class Program
{

    public static void PrintHelperMethod(Print printDel,int val)
    {
        val += 10;
        printDel(val);
    }

    public static void Main(string[] args)
    {
        PrintHelperMethod(delegate(int val) {
            Console.WriteLine("Anonymous method: {0}", val);
        }, 100);
    }
}
```

```
Anonymous method: 110
```

Examples

Delegate & Func<…>

Delegate

```
delegate double MyFunction(double x);
static void Main(string[] args)
{
    MyFunction f = Math.Sin;
    double y = f(4); //y=sin(4)
    f = Math.Exp;
    Console.WriteLine(y);
    y = f(4); //y=exp(4)
    Console.WriteLine(y);
}
```

Create the same with a Func<…>

```
static void Main(string[] args)
{
    double y = 0;
    Func<double, double> myFancyFunc = Math.Sin;
    y = myFancyFunc(4);
    Console.WriteLine(y);
    myFancyFunc = Math.Exp;
    y = myFancyFunc(4);
    Console.WriteLine(y);
}
```

Predicate – returns a bool

- Write a IsEmptyString with a predicate

```
Predicate<string> isEmptyString = String.IsNullOrEmpty;
string s = "Test";
if (isEmptyString(s))
{
    throw new Exception("'Test' cannot be empty");
}
s = "";
if (isEmptyString(s))
{
    throw new Exception("Empty String");
}
```

Action – another special Func

- Write a println Method with an Action<…>

- Know:
  - `Action<string>` is equivalent to `Func<string, void>`

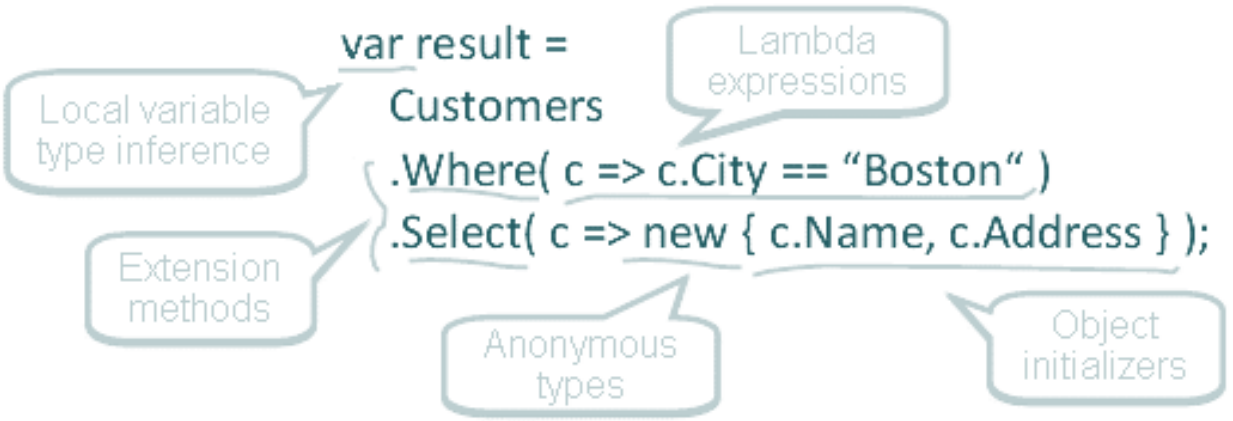
```
Action<string> println = Console.WriteLine;
println("Test");
```

Open DelegateEvent Example.pdf

Implement the following examples:

- Delegate NumberChanger
- Multicast Delegate NumberChanger
- Multicast Delegate Employee
- Delegate with Bookstore

- Happy Birthday – Congrats Event
- Metronome – Tick Event
- Elevator – Warning Event
- School - Fire Alarm Event



Lambda Expression

Anonymos Method - delegate

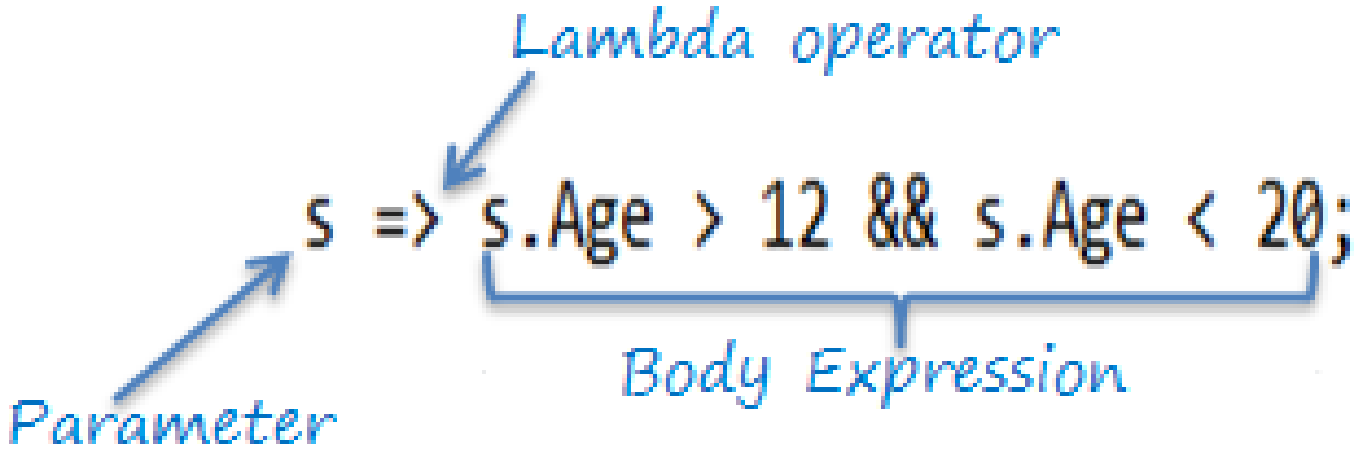
Func, Action, Predicate

Lambda Expression

- is an **anonymous function**
- it's a **method without a declaration**

- That means,
  - no access modifier
  - no return value declaration
  - no name

Lambda Expression



C# =>

- http://www.tutorialsteacher.com/linq/linq-lambda-expression

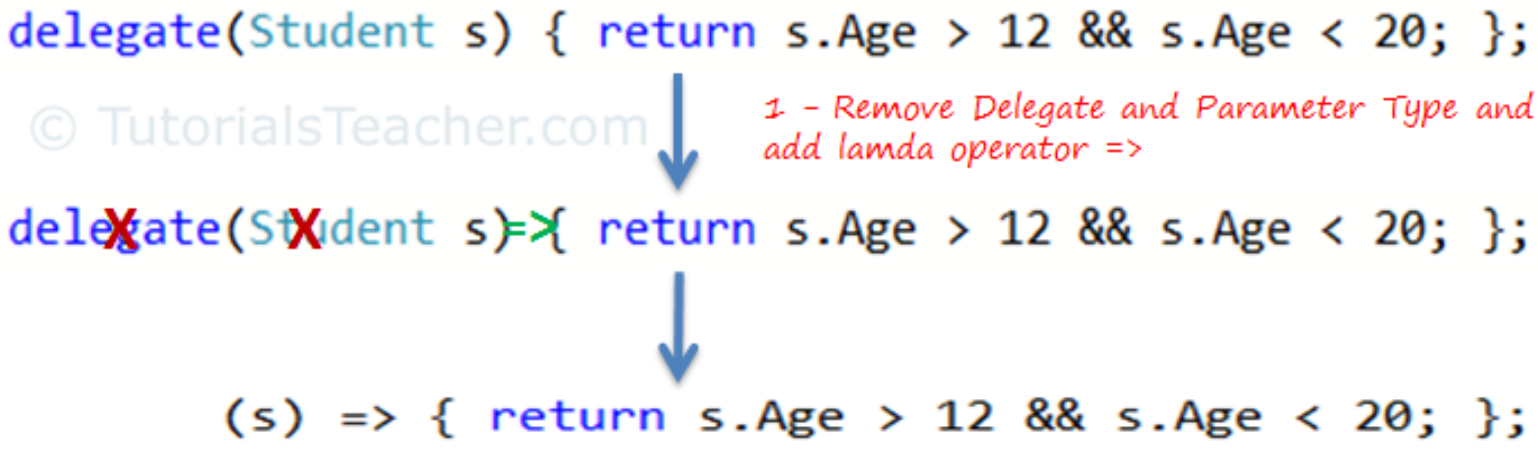
Syntax of Lambda expressions

- *(Eingabeparameter) => {Anweisungsblock}*
- **Parameters => Executed code**

should be short

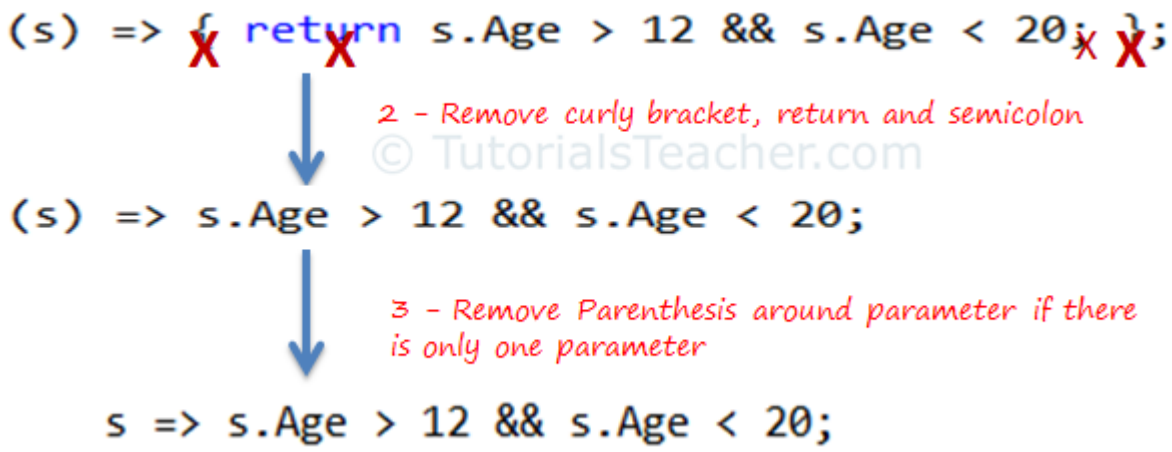
Lambda Expression

- evolves from anonymous method
  - Anonymous Method to Lambda Expression:



Reduce Code

- Omit Curly braces & semicolon



Lambda Expression Examples

- **With multiple parameters**
  - `(s, youngAge) => s.Age >= youngage;`
- **Without any parameters**
  - `() => Console.WriteLine("lambda expression")`
- **Multiple statements in body expression**
  - ```
    (s, youngAge) => {
    Console.WriteLine(“lambda expression ");
    return s.Age >= youngAge; }
    ```
- **Local variable in Lambda Expression body**
  - ```
    s => { int youngAge = 18;
           Console.WriteLine("Lambda expression");
           return s.Age >= youngAge; }
    ```

Lambda Expression

- Rules…
  - if lambda expression do not have parameters empty brackets () should be placed
  - if there is only one parameter in the list brackets are not needed
  - after => sign
    put an expression that will be returned

Lambda Expression Examples

Delegate	Lambda expression
delegate(){ return 3; }	() => 3
delegate(){ return DateTime.Now; }	() => DateTime.Now
delegate(int x){ return x+1; }	(x) => x+1
delegate(int x){ return Math.Log(x+1)-1; }	x => Math.Log(x+1)-1
delegate(int x, int y){ return x+y;}	(x, y) => x+y
delegate(string x, string y){ return x+y;}	(x, y) => x+y

Usefull Advantages

- creating short functions
  - lambda expressions are equivalent to regular functions(delegates)
  - but they are more suitable…

  (x, y) => x + y
  - that can be used both for **adding numbers**
  - but also for **concatenating strings**

    - Using delegates:
    - **need to explicitly define argument types & you cannot use one delegate for other types**

Lambda with Func

```
Func<double, double> f;
f = delegate (double x) { return 3 * x + 1; };
double y = f(4); //y=13
Console.WriteLine(y);

//with Lambda Expression:
```

Add the correct code…

Hello & Good Bye Example

```
static void Hello(string s)
{
    System.Console.WriteLine("  Hello, {0}!", s);
}
static void Goodbye(string s)
{
    System.Console.WriteLine("  Goodbye, {0}!", s);
}
```

```
Action<string> action = Console.WriteLine;
Action<string> hello = Hello;
Action<string> goodbye = Goodbye;
action += Hello;
action += (x) => { Console.WriteLine("  Greating {0} from lambda expression", x); };
action("First"); // called WriteLine, Hello, and lambda expression

action -= hello;
action("Second"); // called WriteLine, and lambda expression

action = Console.WriteLine + goodbye
        + delegate(string x){
                                Console.WriteLine("  Greating {0} from delegate", x);
         };
action("Third");// called WriteLine, Goodbye, and delegate

(action - Goodbye)("Fourth"); // called WriteLine and delegate
```

Do you know the output…

First
  Hello, First!
  Greating First from lambda expression
Second Greating Second from lambda expression
Third
  Goodbye, Third!
  Greating Third from delegate
Fourth
  Greating Fourth from delegate

Write this with Lambda Expression

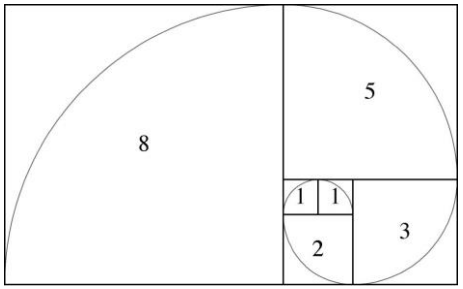
```
static void Main(string[] args) {
    Calc c = new Calc();
    // Die Klasse Calc 'd' stellt
    //die Methoden 'Add','Sub','Mul' und 'Div' zur Verfügung
    // Aufruf ZB: d.Add(x,y)
    int result = 0;
    //ohne lambda
    Func<int, int, int> func;
    func = delegate (int x, int y) { return c.Add(x, y); };
    result = func(3, 1);
    Console.WriteLine(result);
    func = delegate (int x, int y) { return c.Mul(x, y); };
    result = func(7, 3);
    Console.WriteLine(result);
    func = delegate (int x, int y) { return x * y + 4; };
    result = func(3, 4);
    Console.WriteLine(result);
}
```

Solution with Lambda

```
static void Main(string[] args) {
    Calc c = new Calc();
    // Die Klasse Calc 'd' stellt
    //die Methoden 'Add','Sub','Mul' und 'Div' zur Verfügung
    // Aufruf ZB: d.Add(x,y)
    int result = 0;
    //ohne lambda
    Func<int, int, int> func;
    func = delegate (int x, int y) { return c.Add(x, y); };
    result = func(3, 1);
    Console.WriteLine(result);
    func = delegate (int x, int y) { return c.Mul(x, y); };
    result = func(7, 3);
    Console.WriteLine(result);
    func = delegate (int x, int y) { return x * y + 4; };
    result = func(3, 4);
    Console.WriteLine(result);
}
```

```
//lambda
                              lambda;
lambda =
result = lambda(3, 1);
Console.WriteLine(result);
lambda =
result = lambda(7, 3);
Console.WriteLine(result);
lambda =
result = lambda(3, 4);
Console.WriteLine(result);
```

Fibonacci Sequence



Fibonacci relationship

$F_1 = 1$

$F_2 = 1$

$F_3 = 1 + 1 = 2$

$F_4 = 2 + 1 = 3$

$F_5 = 3 + 2 = 5$

In general :

$F_n = F_{n-1} + F_{n-2}$

or

$F_{n+1} = F_n + F_{n-1}$

```
static void Main(string[] args)
{
    int a = 1;
    int b = 1;
    int i = 2;
    int c = a + b;
    Console.WriteLine("Geben Sie eine Zahl ein");
    String input = Console.ReadLine();
    Int32 n = Int32.Parse(input);
    while (i < n)
    {
        c = a + b;
        a = b;
        b = c;
        i = i + 1;
    }
    Console.WriteLine("Die Fibonacci-Zahl von " + n + " ist " + c);

}
```

Solve it recursiv with
Lambda Expression

Convenience

- write a method in the same place you are going to use it
- useful in places where a method is being **used only once**
- and the method **definition is short**

Why lambda expressions?

- Why do we need lambda expressions?
  - Why write a method without a name?

- Reduced typing.
  - No need to specify the name of the function, its return type, and its access modifier.
- No need to look elsewhere
  - When reading the code, you don't need to look elsewhere for the method's definition.

Summary

- is a shorter way of representing anonymous method.
- Syntax: parameters => body expression
  - can have zero parameter.
  - can have multiple parameters in parenthesis ().
  - can have multiple statements in body expression in curly brackets {}
- can be assigned to Func, Action or Predicate delegate
- can be invoked in a similar way to delegate

Lambda Expression & Linq

- A lambda expression is an anonymous function
- it is mostly used to create delegates in Language Integrated Query (LINQ)
  - LINQ:
    is a programming language syntax
    used to query data
  - Another Data Access Technology: ADO.NET