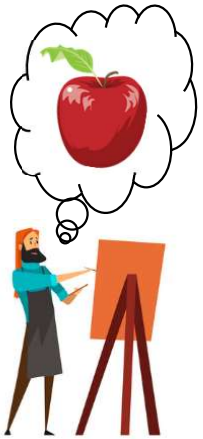
Delegates

Delegates, Lambda, LINQ & Events

www.everyloop.com

Delegates

public Painting Goya(Photo p);



public Painting Leo(Photo p);

public Painting Pico(Photo p);

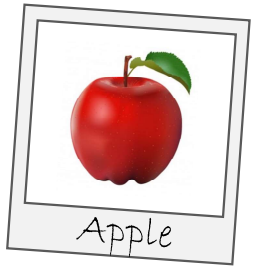
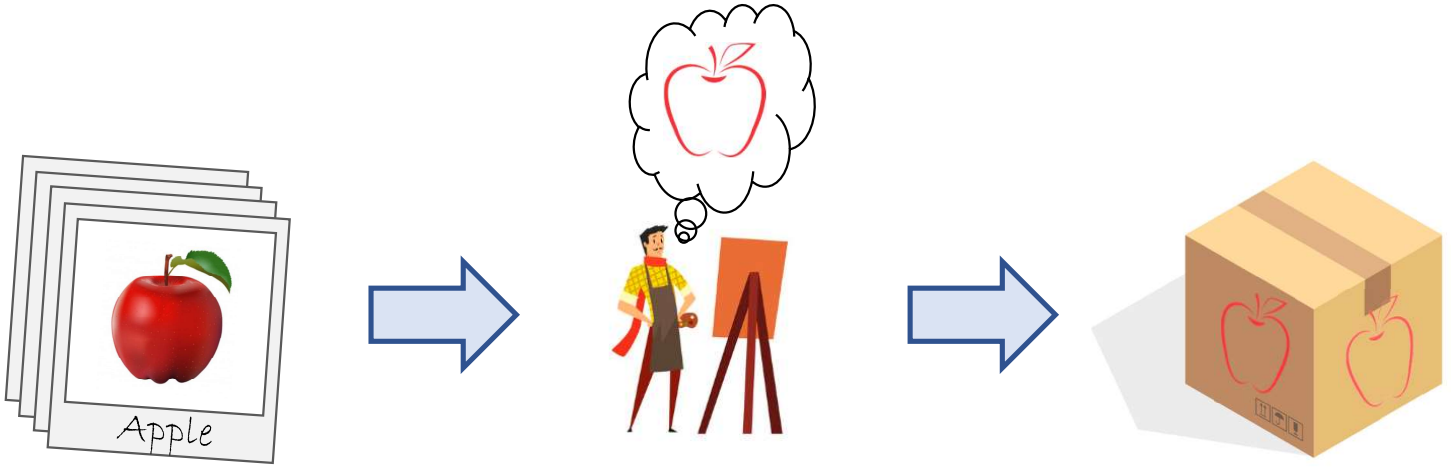


Photo apple = new Photo();

public delegate Painting Artist(Photo p);

Artist hire = new Artist(Pico);

Delegates



public Box DrawAll(Stack<Photo> p, Artist a);

Artist hire = new Artist(Pico);

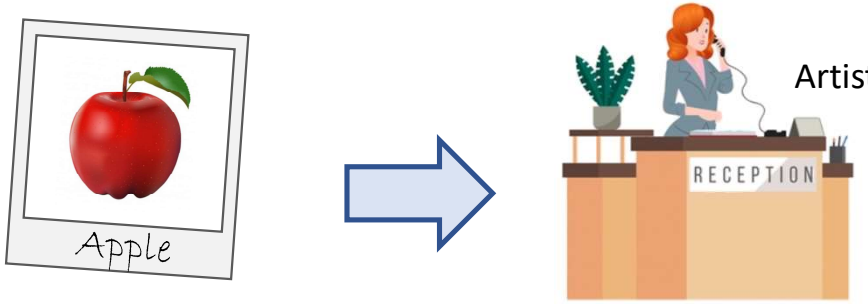


Photo apple = new Photo();

public delegate Painting Artist(Photo p);

Built-in generic delegate types

```
public void DeliverPainting(Photo photo, Painting painting, Action<Photo, Painting> logDelivery)
{
    painting.deliveryAddress = photo.owner.address;
    painting.deliver();

    logDelivery(photo, painting);
}

public Box DrawAll(Stack<Photo> photoStack,  Func<Photo, Painting> artist)
{
    var box = new Box();

    foreach (Photo p in photoStack)
    {
        box.Add(artist(p));
    }
     return box;
}

public void ReturnIfFinished(Photo p, Predicate<Photo> isFinished)
{
    if (isFinished(p)) ReturnPhoto();
}
```

Action

Action<in T1, in T2, ... >

Zero or more input parameters

No output

Func

Func<in T1, in T2, ..., out TResult>

Zero or more input parameters

One output parameter (alway last)

Predicate

bool Predicate<in T>

One input parameter

Output must be bool

Anonymous methods

```
Public Painting Pico(Photo photo)
{
    return createAwesomePainting(photo);
}
```

```
Box picosPaintings = DrawAll(photoStack, Pico);
```

```
Box anonymousPaintings = DrawAll(photoStack,  delegate(Photo photo)  { return createAwesomePainting(photo); });

Box lambdaPaintings = DrawAll(photoStack, p => createAwesomePainting(p));
```

Lambda expressions

Syntax: (input-parameters) => expression

Ex. 1: Lambda expression assigned to delegate

```
Func<int, int> Square = x => x * x;
Action<int> PrintInt = x => Console.WriteLine(x);
PrintInt(Square(3));    // Prints: 9
```

Ex. 2: Lambda expression with multiple parameters

```
(person, legalAge) => person.Age >= legalAge
(a, b, c) => $"{a} + {b} + {c} = {a+b+c}"
(r, h) => r * r * Math.PI * h
```

Ex. 3: Lambda expression without parameters

```
() => Console.Writeline("Hello world!")
() => SaveFile()
```

Ex. 4: Multiple statements in lambda expression body

```
(photo, painting) =>
{
    painting.deliveryAddress = photo.owner.address;
    return painting;
}
```

Anonymous types

a type (class) without any name that can contain public read-only properties only.
It cannot contain other members, such as fields, methods, events, etc.

```
var anonymous = new { x = 3, y = 5.4f, z = 45L, s = "test" };
```

Array of anonymous types

```
var people = new[] {
    new { LastName = "Eriksson",  FirstName = "Anders", Age = 39 },
    new { LastName = "Palm", FirstName = "Lisa",  Age = 40 },
    new { LastName = "Andersson",  FirstName = "Per",  Age = 31 },
    new { LastName = "Lundberg",  FirstName = "Anna",  Age = 66 },
    new { LastName = "Eriksson",  FirstName = "Camilla",  Age = 65 },
};
```

Nested anonymous type

```
var person = new
{
    name = "Anders Eriksson",
    age = 45,
    contactInfo = new { eMail = "anders@gmail.com", phone = "07023485734" }
};
```

Language Integrated Query (LINQ)

A .NET component adding data querying capabilities to .NET languages,
partly by providing extension methods on the IEnumerable interface
for standard query operators like filtering, sorting, grouping and aggregation.

Extension methods

Where<*T>(Func<T, bool> predicate)*
Filters collection based on the given critera.

OfType<*TResult>()*
Filters collection based on the given type.

GroupBy<*TSource, TKey>(Func<TSource, TKey> keySelector)*
Grouping data based on key value.

Select<*TSource, TRresult>(Func<TSource, TResult> selector)*
Get value from collection.

OrderBy<*TSource, TKey>(Func<TSource, TKey> keySelector)*
Sorts collection on the specified field.

OrderByDescending<*TSource, TKey>(Func<TSource, TKey> keySelector)*
Sorts collection on the specified field in descending order.

ThenBy<*TSource, TKey>(Func<TSource, TKey> keySelector)*
Second level sorting in ascending order.

ThenByDescending<*TSource, TKey>(Func<TSource, TKey> keySelector)*
Second level sorting in descending order.

Examples

Ex 1: people.Where(p => p.Age >= 18).OrderBy(p => p.LastName).ThenBy(p => p.FirstName).Select(p => new {p.LastName, p.FirstName})

Ex 2: people.GroupBy(p => p.LastName).Select(p => new {lastName = p.Key, averageAge = p.Average(p => p.Age), count = p.Count()})

LINQ – Query syntax

With LINQ Microsoft introduced certain keywords to C#, making it possible to write type-checked SQL-like queries integrated directly in the language. LINQ queries can be used on various sources, like XML, RDBMS, and ADO.NET

from string in listOfStrings where string.Length == 2 select string

iterator collection selection projection

Keywords

orderby <keySelector> ascending/descending goes between selection and projection.

Ending query with group <iterator> by <keySelector> returns a list of lists.

group <iterator> by <keySelector> into <groupName> followed by select.

Examples

Ex 1: from p in people where p.Age >= 18 orderby p.LastName, p.FirstName select new {p.LastName, p.FirstName}

Ex 2: from p in people group p by p.LastName into g select new {lastName = g.Key, averageAge = g.Average(g => g.Age), count = g.Count()}

Yield and deferred execution

In LINQ the execution of quieres is deferred until the realized value is actually required, unless immediate execution is forced by aggregation functions, ToList() or similar methods.

Deferred execution is implemented using the *yield* keyword. Using *yield* we indicate that the method in which it appears is an iterator, and the *yield return* statement is used to return each element one at a time, until a *yield break* or end of method is reached.

IEnumerable GetNumbers() { yield return 1; yield return 2; yield return 3; }

Returns an IEnumerable which gets called each iteration of a foreach loop. Execution resumes after each yield return.

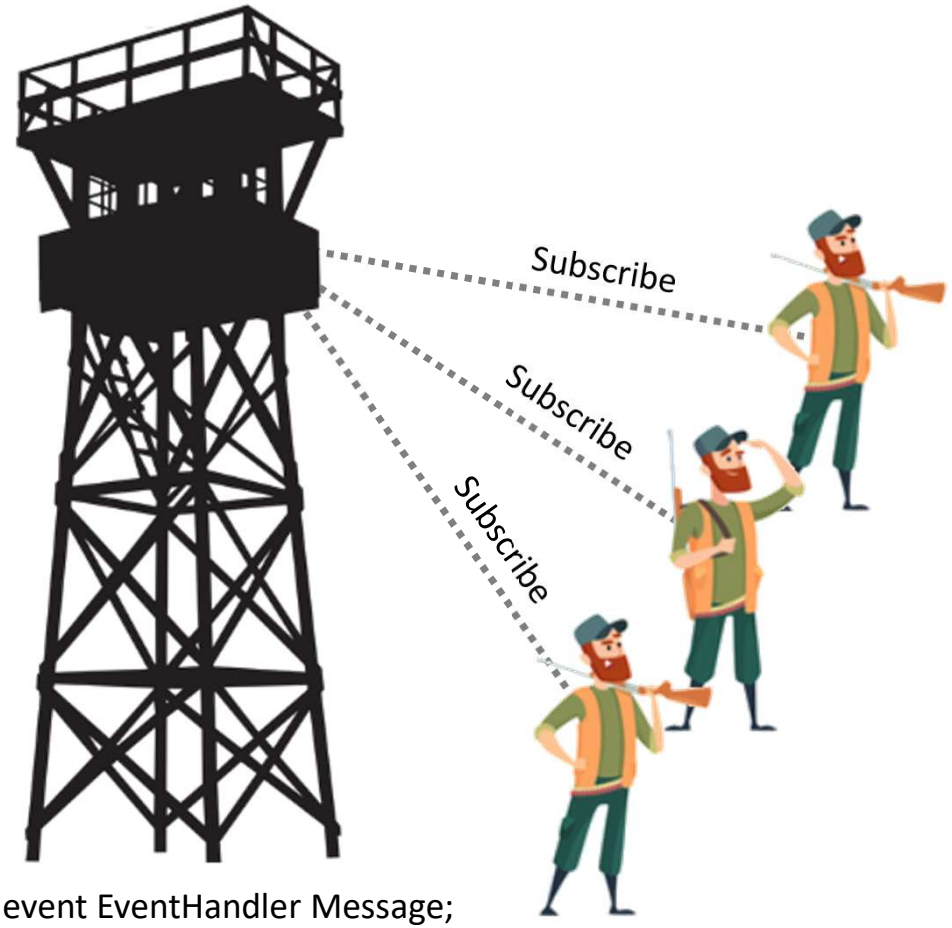
foreach(var number in GetNumbers()) Console.WriteLine(number);

```
IEnumerable<int> GenerateWithoutYield()
{
    var i = 0;
    var list = new List<int>();
    while (i < 5) list.Add(++i);
    return list;
}
```

```
IEnumerable<int> GenerateWithYield()
{
    var i = 0;
    while (I < 5) yield return ++i;
}
```

Events

Message?.Invoke(this, aim);
Message?.Invoke(this, shoot);



Public event EventHandler Message;

Events are used to enable a class or object to notify other classes or objects about the action that is going to happen

To declare an event we use the *event* keyword with delegate type.

The class that sends (or raises) the event is called a publisher.
It is the publisher that determines when an event is raised.

The classes that recives (or handle) the event are called subscribers.
A subscriber determines what action to take in response to an event.

An event can have multiple subscribers, and a subscriber can handle multiple events from multiple publishers.

By using += operator we can subscribe to an event.

By using -= operator we can unsubscribe to an event.

Before raising an event we need to check if it's subscribed or not.

To raise an event we need to invoke the event delegate.

Events are based on EventHandler delegate and EventArgs base class.