Collections, Delegates and Events

oxygen

19. september 2017

Agenda

oxygen

Collections in C#

Delegates

Lambda Expressions

Events

Features and generics

Features

oxygen

Define sort order of objects

- Implement IComparable<T>
- Create a comparer

Define an indexer

- Allows access to Collections like arrays
- Indexer key can be string or integer

Iterate with Foreach-loop

- Implement IEnumerable<T>

Generics in C#

oxygen

What do you know about generics?

Are generics only related to Collections?

Generics in C# (cont.)

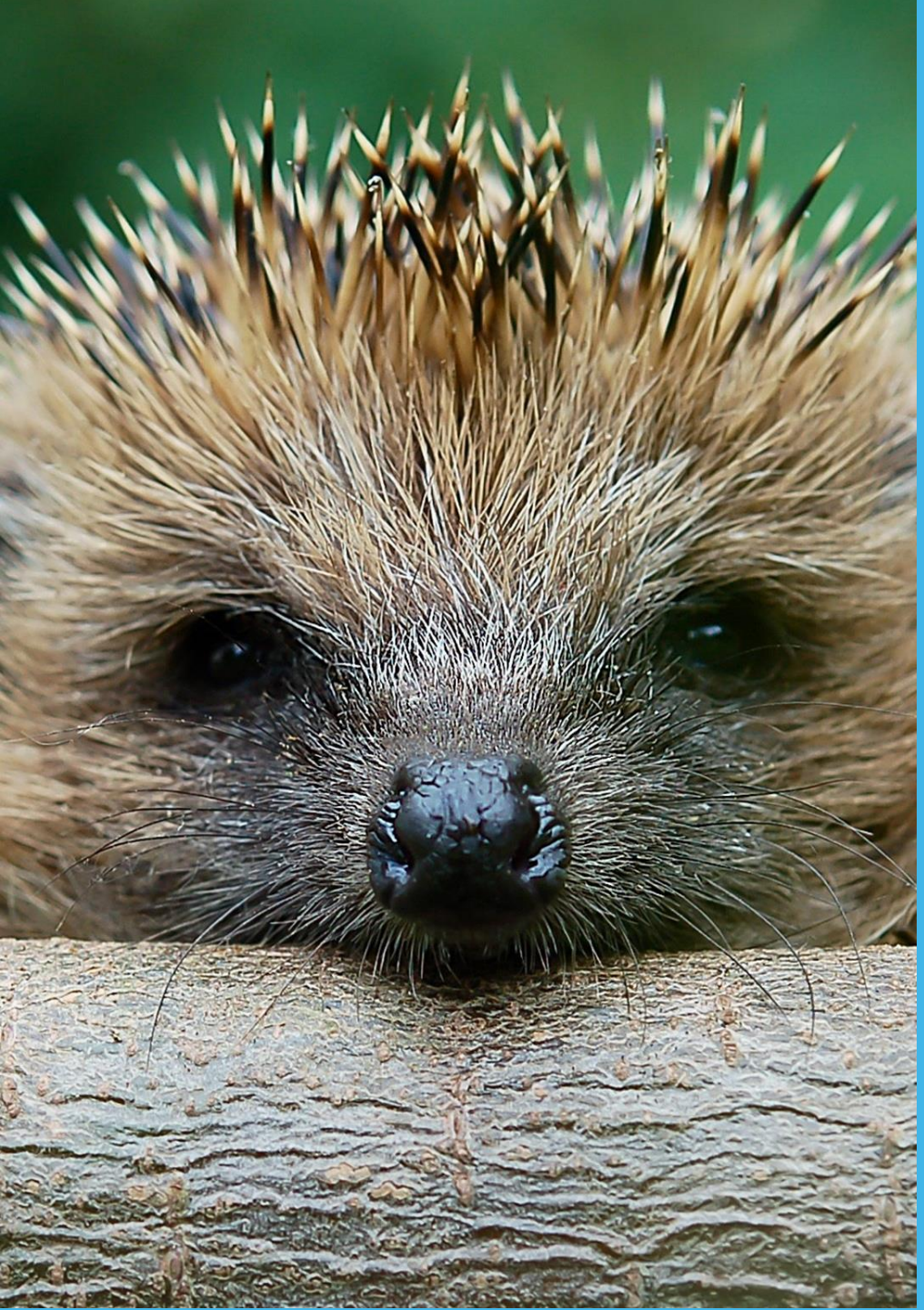
oxygen

Syntax:

- class MyClass <T, K>
- T Method(T)
- Delegate<T>

Customizing generics

- : BaseClass, Interface, class (reference)
- class MyClass <T: MyInterface>

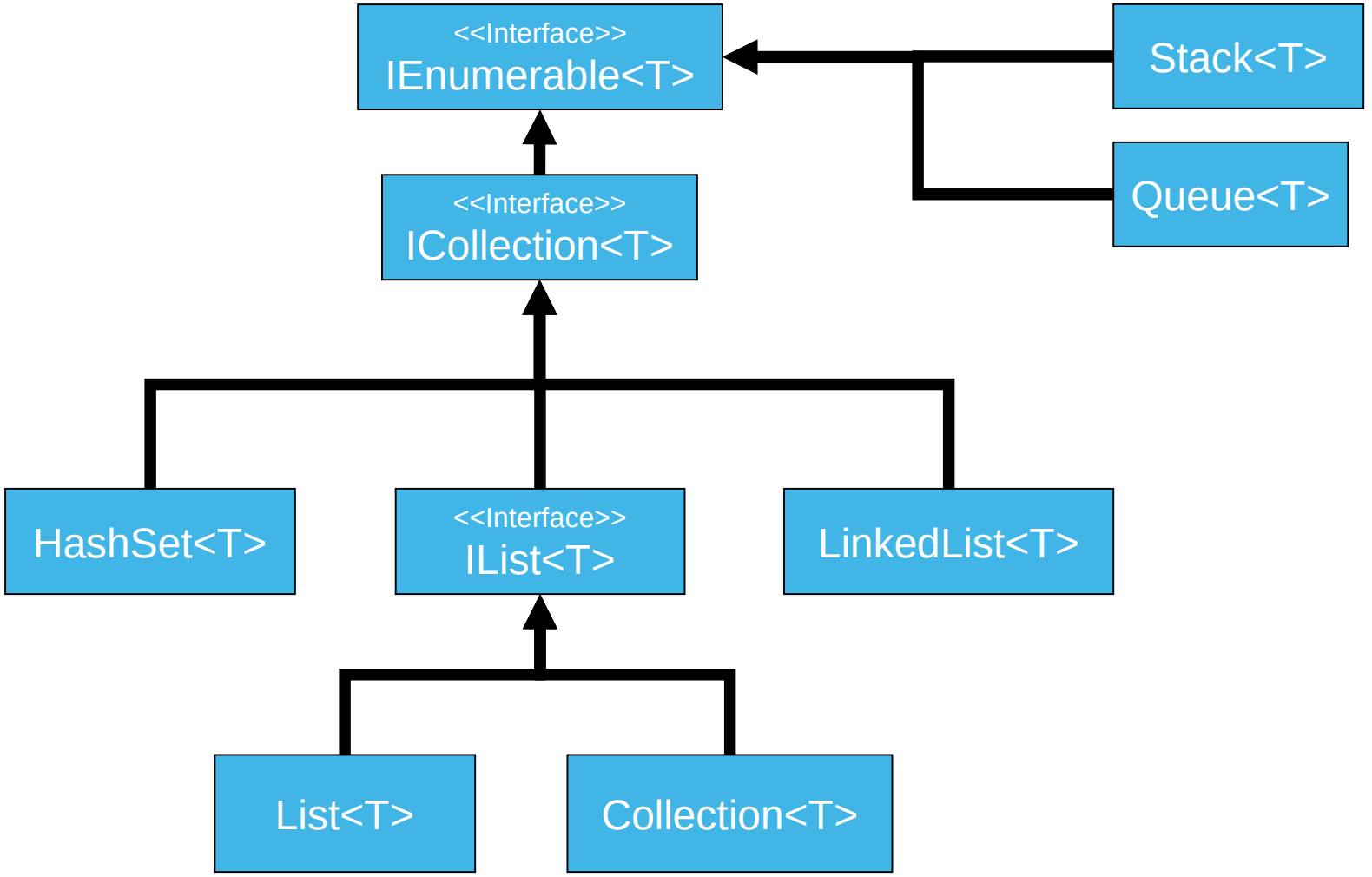


oxygen

Collections hierarchy

Generic Hierarchy

oxygen



Assignment in class

oxygen

Look up a few different collections in C# .NET

Find out:

- What are they meant for?
- Any important members?
- How to use them

Examples: (Find some yourself)

1. System.Collection.BitArray
2. System.Collection.Specialized.BitVector32
3. System.Collection.Generic.HashSet<T>
4. System.Collection.Generic.Stack<T>
5. System.Collection.Generic.Sorted<T>

Interfaces

Interfaces

oxygen

IEnumerable<T>

ICollection<T>

IList<T>

IEnumerable<T>

oxygen

Implementing this interface allows to iterate through all elements in the collection, using foreach-loop

```
Interface IEnumerable<T>{
   IEnumerator GetNumerator();
   IEnumerator<T> GetNumerator();
}
```

```
Interface IEnumerator<T>{
    Object Current {get; }; // Get current object
    T Current {get; };        // Get current T
    bool MoveNext();       // Go to the next object
    void Reset();             // Go back to the first
}
```

ICollection<T>

oxygen

Base interface for all collections

```
Interface ICollection <T>: IEnumerable\<T>{
    int Count {get};
    bool IsSynchonized {get};
    object SyncRoot {get};
    void Add(T item);
    void Clear()
    bool Contains(T item);
    bool Remove(T item)
    void CopyTo(T [] a, int index);
}
```

IList<T>

oxygen

Interface that specifies that it is possible to access specific elements using an index

```
Interface IList<T>{
   T this[int index]
   int IndexOf(T element)
   void Insert(int index, T element)
   void RemoveAt(int index)
}
```

Therefore this is possible:

```
IList<string> myList = new List<string>();
myList.Add(“Steffen”);
myList.Add(“Peter”);
var firstName = myList[0];
myList[1] = “Peder”;
```

Classes

Collection<T>

oxygen

Most interfaces and classes discussed in this chapter belong to the namespace System.Collections.Generic

Collection<T> is for some reason part of System.Collections.ObjectModel

Collection<T> implements the IList<T> interface

List<T> implements the same interface, Collection<T> is a minimal implementation

List<T> has much more functionality.

Allows for indexed access, but has no size limit (Arrays does)

Collection<T> (cont.)

oxygen

Initialization:

Collection<int> col = new Collection<int>();

Collection<int> col = new Collection<int>(new int[] {1,2,3,4});

Collection<int> col = new Collection<int>{1,2,3,4};

Last one is refered to as a Collection Initializer.
A collection initializer uses the Add method repeatedly to insert the elements within {...} into an empty list.

Assignment: Try working with the Collection<T> in a console application.

List<T>

oxygen

Collection<T> and List<T> implement the same interface

List<T> offers many more operations

Compared with Collection<T> the class List<T> offers sorting, searching, reversing, and conversion operations.

Examples:

- RemoveAll(Predicate<T>)
- AddRange(IEnumerable<T>)
- Exists(Predicate<T>)

Exercise: Find differences on Collection<T> and List<T>

Try changing from Collection to List and use some of the special operations

LinkedList<T>

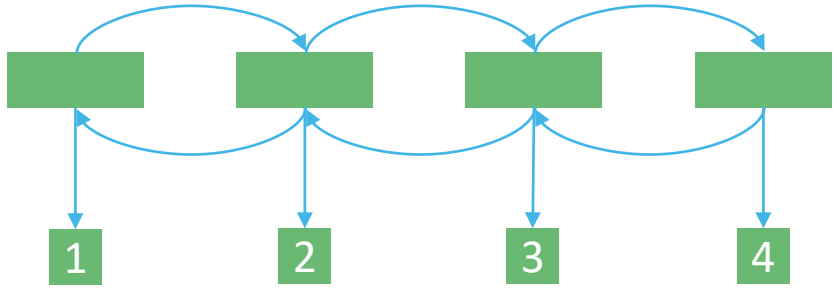
oxygen

Both Collection<T> and List<T> where based on arrays

Now we focus on a list type that is based on a linked representation

The generic class LinkedList<T> relies on a "building block class" LinkedListNode<T>

LinkedList<T> does not implement indexed access, as of IList<T>



Time Complexity

oxygen

Operation	Collection<T>	List<T>	LinkedList<T>
this[i]	O(1)	O(1)	-
Count	O(1)	O(1)	O(1)
Add(e)	O(1) or O(n)	O(1) or O(n)	O(1)
Insert(i,e)	O(n)	O(n)	-
Remove(e)	O(n)	O(n)	O(n)
IndexOf(e)	O(n)	O(n)	-
Contains(e)	O(n)	O(n)	O(n)
BinarySearch(e)	-	O(log n)	-
Sort()	-	O(n log n) or $O(n^2)$	-
AddBefore(lln)	-	-	O(1)
AddAfter(lln,e)	-	-	O(1)
Remove(lln)	-	-	O(1)
RemoveFirst()	-	-	O(1)
RemoveLast()	-	-	O(1)

Use the interface

oxygen

For any given collection in your code, it is recommended to use interfaces throughout the code

I.e. use IList instead of List

It is an advantage to use collections via interfaces instead of classes

If possible, only use collection classes in instantiations, just after new

This leads to programs with fewer bindings to concrete implementations of collections

With this approach, it is easy to replace a collection class with another

Program against collection interfaces, not collection classes

oxygen

Dictionaries

What is a dictionary

oxygen

A dictionary is a data structure that maps keys to values.

A given key can have at most one value in the dictionary.

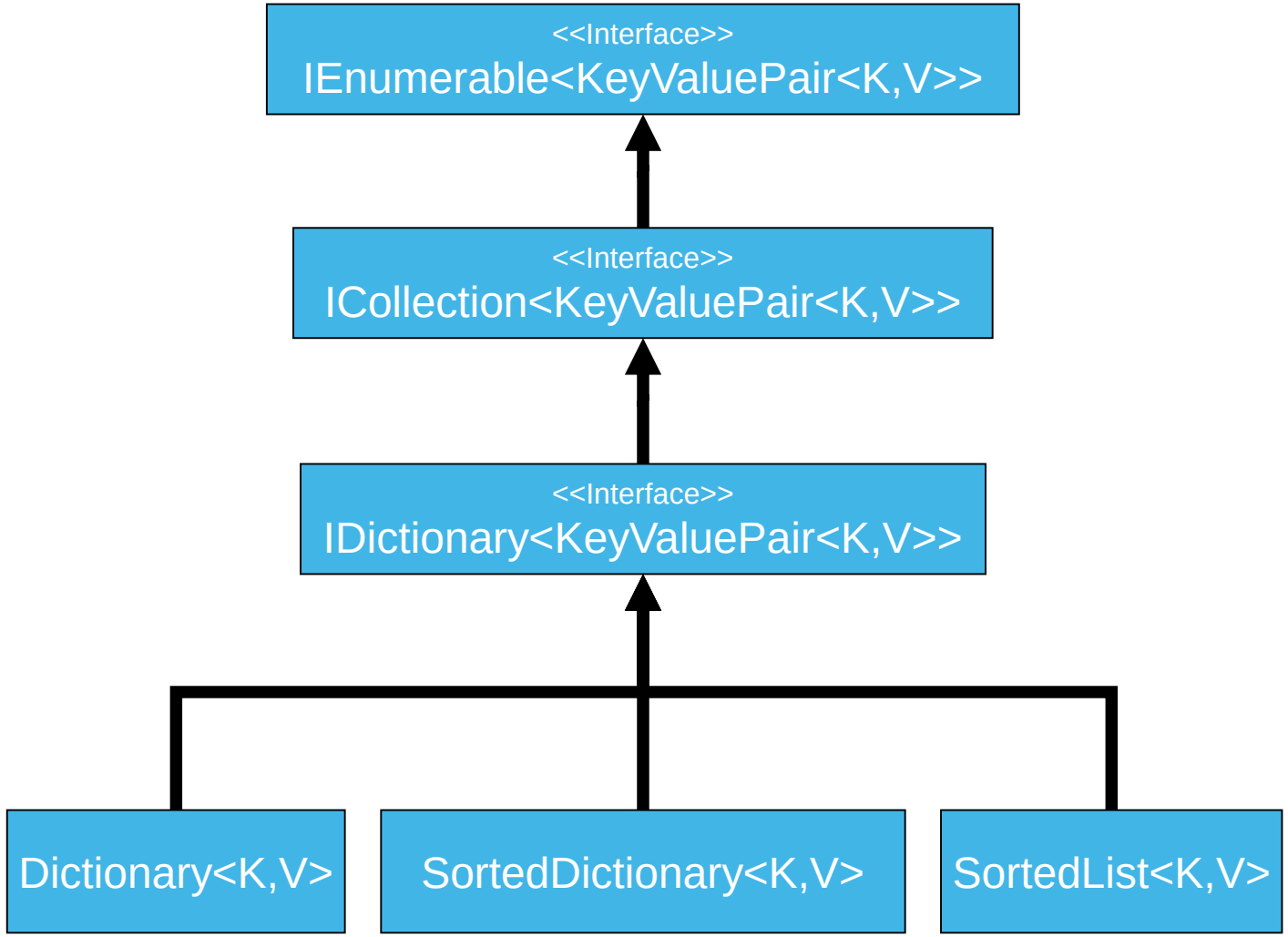
In other words, the key of a key-value pair must be unique in the dictionary.

A given value can be associated with many different keys.

Gives good performance in regards to lookup

Generic Hierarchy

oxygen



Generic Hierarchy (cont.)

oxygen

Dictionary<K,V> is implemented in terms of a hashtable that maps objects of type K to objects of type V.

SortedDictinonary<K,V> relies on binary search trees.

SortedList<K,V> is based on a sorted arrays.

Interfaces

IDictionary<K,V>

oxygen

Is a sub-interface of ICollection<KeyValuePair<K,V>>

Therefore we know that Add, Contains and Remove are present

- V this[K key] - both getter and setter; the setter adds or mutates
- void Add(K key, V value) - only possible if key is not already present
- bool Remove(K key)
- bool ContainsKey(K key)
- bool TryGetValue(K key, out V value)
- ICollection<K>Keys - getter
- ICollection<V>Values - getter

Time complexity

oxygen

Operation	Dictionary<K,V>	SortedDictionary<K,V>	SortedList<K,V>
this[key]	O(1)	O(log n)	O(log n) or O(n)
Add(key,value)	O(1) or O(n)	O(log n)	O(n)
Remove(key)	O(1)	O(log n)	O(n)
ContainsKey(key)	O(1)	O(log n)	O(log n)
ContainsValue(value)	O(n)	O(n)	O(n)

Delegates

oxygen

A delegate is a type the values of which consist of methods

Delegates allow us to work with variables and parameters that contain methods

Delegates in C#

oxygen

Seen in relation to similar, previous object-oriented programming languages (such as Java and C++) this is a new topic.

The inspiration comes from functional programming where functions are *first class values*.

If *x* is a first class value *x* can be passed as parameter, *x* can be returned from functions, and *x* can be part of data structures.

With the introduction of delegates, methods become first class values in C#.

Delegates in C# (cont.)

oxygen

Thus, a delegate in C# defines a type, in the same way as a class defines a type.

A delegate reflects the signature of a set of methods, not including the method names, however.

A delegate is a reference type in C#. It means that values of a delegate type are accessed via references, in the same way as an object of a class always is accessed via a reference.

In particular, null is a possible delegate value.

What we have shown above gives you the flavor of functional programming.

In functional programming we often generate new functions based on existing functions

Delegates in C# (cont.)

oxygen

Delegates make it possible to approach the functional programming style

Methods can be passed as parameters to, and returned as results from other methods

Exercise

oxygen

We are gonna work with sorting an array of points, fetch Point class on BB

Sort the array of points by use of one of the static Sort methods in System.Array

We will use the one that takes a Comparison delegate, Comparison<T>, as the second parameter. Please find this method in your documentation browser.

Why do we need to pass a Comparison predicate to the Sort method?

Comparison<Point> is a delegate that compares two points, say p1 and p2. Pass an actual delegate parameter to Sort in which

p1 <= p2 if and only if p1.X + p1.Y <= p2.X + p2.Y

https://pastebin.com/GnvxiCXx

Lambda Expressions

Lambda expressions

oxygen

A lambda expression is a value in a delegate type.

The notation of lambda expression adds some extra convenience to the notation of delegates.

We simplify the syntax:

delegate(formal-parameters){body}

To:

Formal-parameters => body

Going from delegate to lambda

oxygen

```
NumericFunction[] equivalentFunctions =
        new NumericFunction[]{
          delegate (double d){return d*d*d;},
          (double d) => {return d*d*d;},
          (double d) => d*d*d,
          (d) => d*d*d,
          d => d*d*d
       };
```

Lambda in relation to delegates

oxygen

- The body can be a statement block or an expression
- Uses the operator => which has low priority and is right associative
- May involve implicit inference of parameter types
- Lambda expressions serve as syntactic sugar for a delegate expression

Events

Events

oxygen

Core concept of Event driven programming

In command-driven programming, the computer prompts the user for input. The program stops and waits. When a command is issued by the user, the program is continued. The program will analyze the command and carry out an appropriate action.

In event-driven programming the program reacts on *what happens on the elements of the user interface*, or more generally, *what happens on some selected state of the program*. When a given event is triggered the actions that are associated with this particular event are carried out.

Inversion of control

Don't call us - we call you

Operations vs events

Operations

- Belongs to a class
- Is called explicitly - directly or indirectly - by other operations

Events

- Belongs to a class
- Contains one or more operations, which are called when the event is triggered.
- The operations in the event are called implicitly

oxygen

Events in C#

Events in C#

oxygen

In C# an event in some class C is a variable of a delegate type in C.

Like classes, delegates are reference types.

This implies that an event holds a reference to an instance of a delegate.

The delegate is allocated on the heap.

Events in C# (cont.)

oxygen

From inside some class, an event is a variable of a delegate type.

From outside a class, it is only possible to add to or remove from an event.

Events are intended to provide notifications, typically in relation to graphical user interfaces.

- An event can only be activated from within the class to which the event belongs
- From outside the class it is only possible to add (with +=) or subtract (with -=) operations to an event.
- It is not possible to 'reset' the event with an ordinary assignment

oxygen

Questions?