The final session

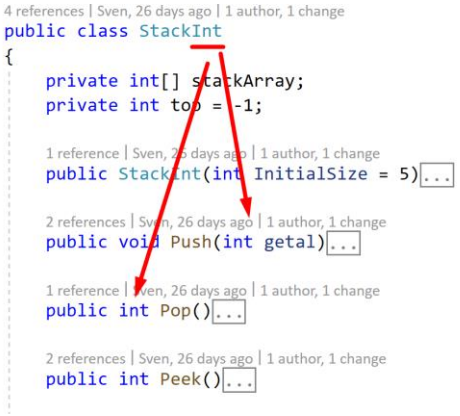
Bachelor IT
Sven Mariën
(sven.marien01@ap.be)

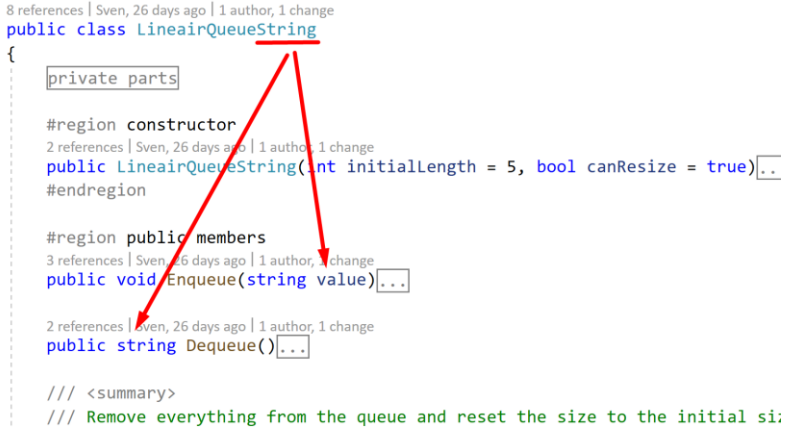
AP HOGESCHOOL ANTWERPEN

AP.BE

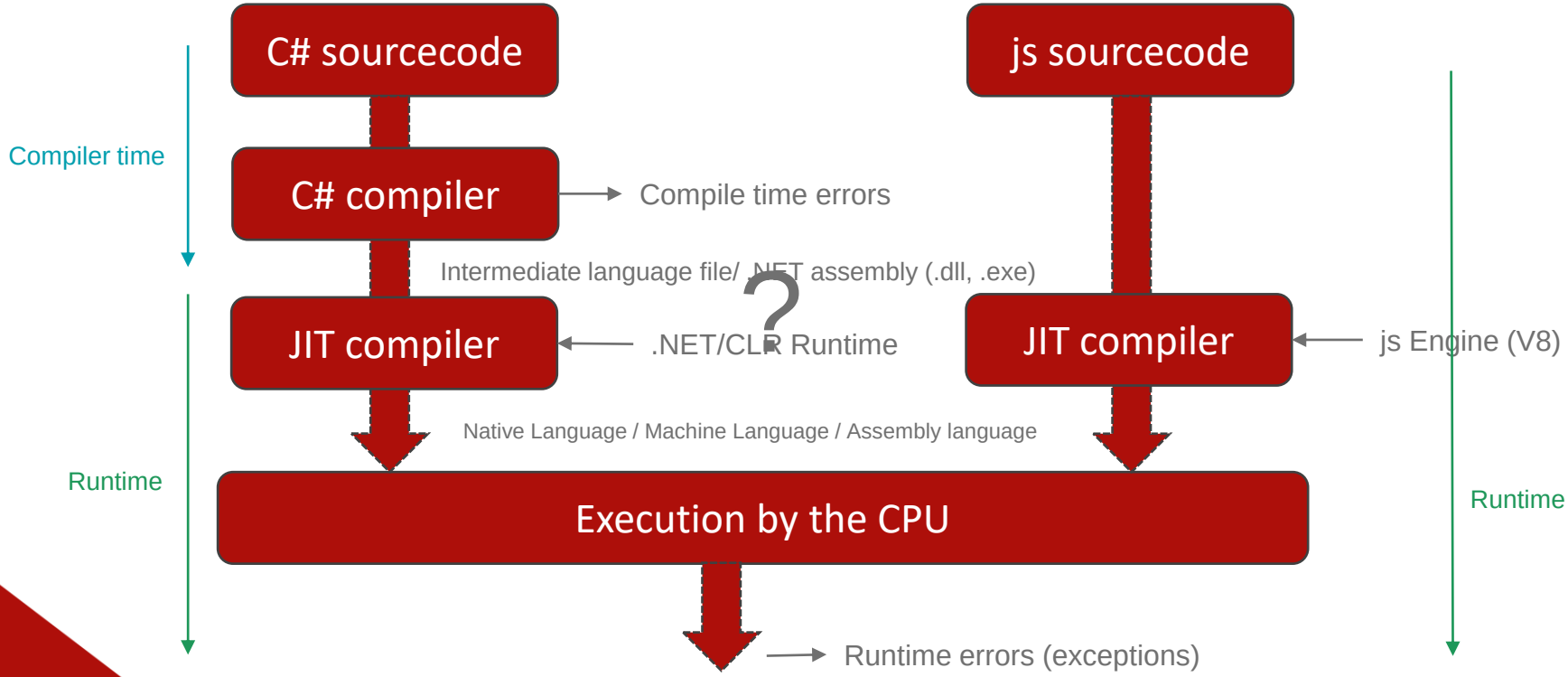
Datastructures

- Queue & Stack
- Sorteeralgoritmen
- Linked List & Tree
- Zoekalgoritmen



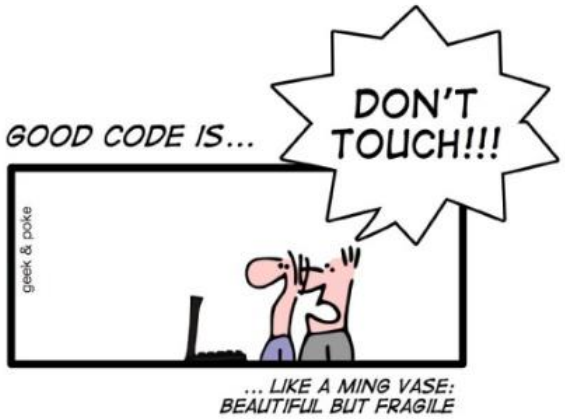


C# versus javascript



Compiler of geen compiler…

- **Compiler errors** brengen een probleem **onmiddellijk** aan het licht tegenover **run-time exceptions** die pas optreden **als de code** daadwerkelijk **wordt uitgevoerd**.
- Beeld je in dat je een applicatie hebt met 10.000 lijnen broncode, dwz. dat je elke lijn steeds moet testen als je iets hebt aangepast in de code.



C# en js: verschil in typesystems

- C# werkt
  - met een “**static type system**”
  - dwz. dat **elke variabele of parameter** een **vast type** heeft.
  - Dat type ligt vast van bij de **declaratie**
- js werkt
  - met een “**dynamic type system**”
  - dwz. dat het **type** van een variable **kan wijzigen** tijdens de levensduur.
  - Het **type** kan zelfs “**undefined**” zijn…

Types: c# vs. javascript

C#

```
int getal = 10;
string woord = "hallo";
bool vlag = true;
var teller = 29;        //Type inference (afleiden van het type adhv. de initële waarde)

getal = "hallo";        //this is not js ..
```

js

```
var a;
console.log(`a has value:${a} & is now type: '${typeof(a)}'`);
a = 10
console.log(`a has value:${a} & is now type: '${typeof(a)}'`);
a = "DATASTRUCTURES"
console.log(`a has value:${a} & is now type: '${typeof(a)}'`);
a = true
console.log(`a has value:${a} & is now type: '${typeof(a)}'`);
```

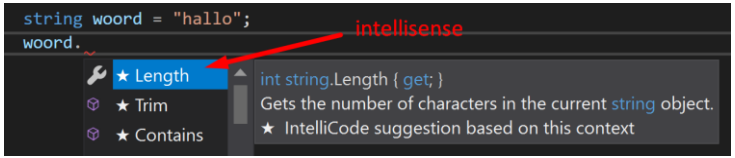
```
a has value:undefined & is now type: 'undefined'
a has value:10 & is now type: 'number'
a has value:DATASTRUCTURES & is now type: 'string'
a has value:true & is now type: 'boolean'
```

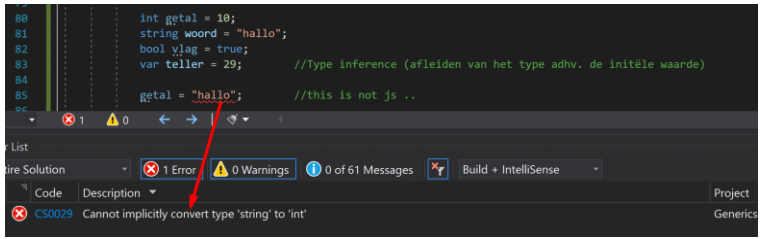
C#

```
object watzouikhiermeedoen = "Help";

dynamic weGaanDeJavascriptToerOp = 33;
weGaanDeJavascriptToerOp = "Dit kan dus echt ook in c# :-)";
```

Static type system

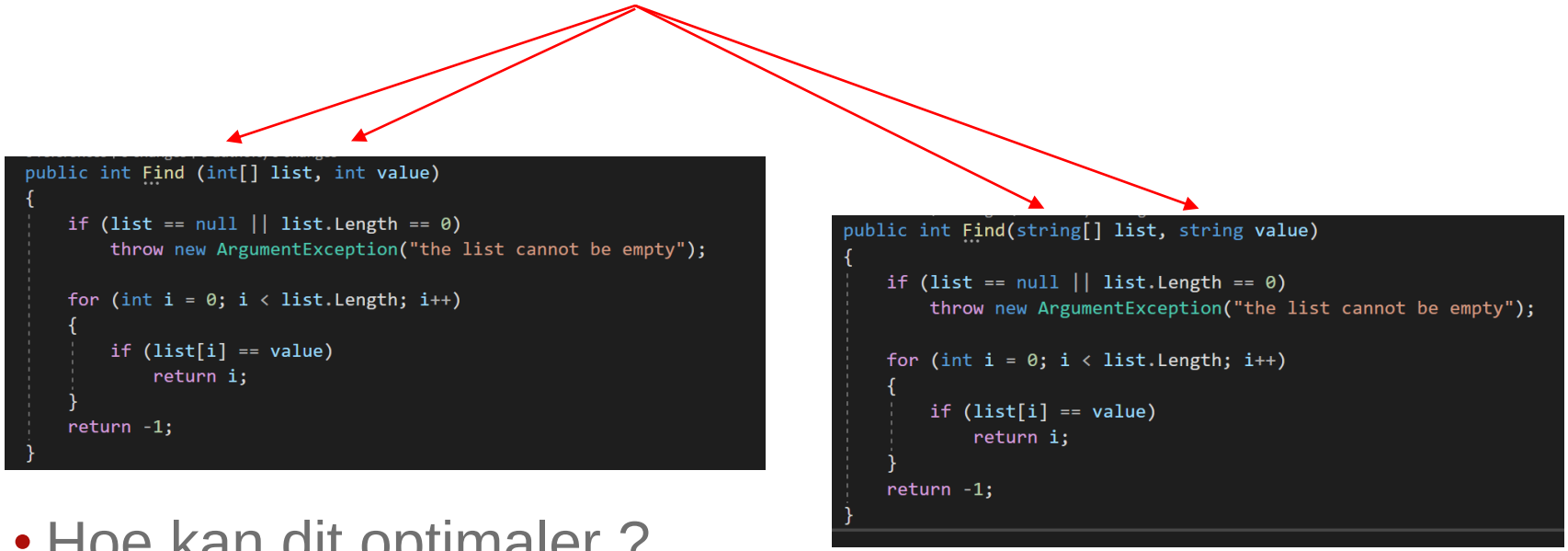
- Werken met static types:
  - **Intellisense:** Je weet “at development time” exact welke properties en methodes je kan aanroepen.
  - **Type safety:** Compiler zal fouten geven als je de applicatie compileert (opstart).





c#: Maximaal broncode hergebruiken

- Als we code vergelijken voor bv. een **lineair search**:



```
public int Find (int[] list, int value)
{
    if (list == null || list.Length == 0)
        throw new ArgumentException("the list cannot be empty");

    for (int i = 0; i < list.Length; i++)
    {
        if (list[i] == value)
            return i;
    }
    return -1;
}
```

```
public int Find(string[] list, string value)
{
    if (list == null || list.Length == 0)
        throw new ArgumentException("the list cannot be empty");

    for (int i = 0; i < list.Length; i++)
    {
        if (list[i] == value)
            return i;
    }
    return -1;
}
```

- Hoe kan dit optimaler ?

Broncode hergebruiken => type object ?

- Laten we starten met een eenvoudigere methode:

```
public int[] CreateArray(int item1, int item2)
{
    int[] array = new int[2];
    array[0] = item1;
    array[1] = item2;
    return array;
}
```

```
public string[] CreateArray(string item1, string item2)
{
    string[] array = new string[2];
    array[0] = item1;
    array[1] = item2;
    return array;
}
```

?

```
public object[] CreateArray(object item1, object item2)
{
    object[] array = new object[2];
    array[0] = item1;
    array[1] = item2;
    return array;
}
```

```
string a = "AP";
string b = "Hogeschool";

Test t = new Test();
object[] result = t.CreateArray(a, b);

if (result[1].Length > 5)
{
    //...
}
```

Compiler error

CS1061: 'object' does not contain a definition for 'Length' and no accessible extension method 'Length' accepting a first argument of type 'object' could be found (are you missing a using directive or an assembly reference?)

Broncode hergebruiken => type dynamic ?

```
public int[] CreateArray(int item1, int item2)
{
    int[] array = new int[2];
    array[0] = item1;
    array[1] = item2;
    return array;
}
```

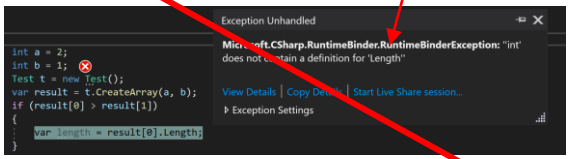
```
public string[] CreateArray(string item1, string item2)
{
    string[] array = new string[2];
    array[0] = item1;
    array[1] = item2;
    return array;
}
```

?

```
public dynamic[] CreateArray(dynamic item1, dynamic item2)
{
    dynamic[] array = new dynamic[2];
    array[0] = item1;
    array[1] = item2;
    return array;
}
```

```
int a = 1;
int b = 2;
Test t = new Test();
var result = t.CreateArray(a, b);
if (result[0] > result[1])
{
    var length = result[0].Length;
}
```

Runtime error



Optie 3: **Generics** !

- We schrijven de code 1x
- De code kan worden hergebruikt voor meerdere types
- De performantie is beter (geen boxing/unboxing)
- De types toch ook worden “at compile time” geverifieerd.
- We hebben dus ook intellisense !

Use generic types to maximize code reuse, type safety and performance !

Generics: generieke methode maken

```
public int[] CreateArray(int item1, int item2)
{
    int[] array = new int[2];
    array[0] = item1;
    array[1] = item2;
    return array;
}
```

```
public string[] CreateArray(string item1, string item2)
{
    string[] array = new string[2];
    array[0] = item1;
    array[1] = item2;
    return array;
}
```

?

```
public T[] CreateArray<T>(T item1, T item2)
{
    T[] array = new T[2];
    array[0] = item1;
    array[1] = item2;
    return array;
}
```

```
int a = 2;
int b = 1;
Test t = new Test();
var result = t.CreateArray<int>(a, b);
if (result[0] > result[1])
{
    var length = result[0].Length;
}
```

Generics: generieke methode aanroepen

```
public T[] CreateArray<T>(T item1, T item2)
{
    T[] array = new T[2];
    array[0] = item1;
    array[1] = item2;
    return array;
}
```

```
int a = 2;
int b = 1;
Test t = new Test();
var result = t.CreateArray<int>(a, b);
if (result[0] > result[1])
{
    var length = result[0].Length;
}
```

```
string a = "Hallo";
string b = "AP";
Test t = new Test();
var result = t.CreateArray<string>(a, b);
if (result[0] > result[1])
{
    var length = result[0].Length;
}
```

```
Auto auto = new Auto()
{
    AantalKm = 10000,
    Bouwjaar = 2005,
    Brandstof = Brandstof.Waterstof,
    Kleur = "Rood",
    Model = "Ferrari F40"
};

Auto auto2 = new Auto()
{
    AantalKm = 20000,
    Bouwjaar = 2005,
    Brandstof = Brandstof.Electriciteit,
    Kleur = "Groen",
    Model = "Ferrari F40"
};

Test t = new Test();
var result = t.CreateArray<Auto>(auto, auto2);
if (result[1].B)
```

Bouwjaar
Brandstof
int Auto.Bouwjaar { g

Compile time type checking

Generic Stack bouwen

```
public class StackInt
{
    private int[] stackArray;
    private int top = -1;

    1 reference | Sven, 13 days ago | 1 author, 1 change
    public StackInt(int InitialSize = 5)
    {
        stackArray = new int[InitialSize];
    }

    2 references | Sven, 13 days ago | 1 author, 1 change
    public void Push(int getal)
    {
        if (IsFull)
            throw new Exception("Sorry the stack is full");

        stackArray[++top] = getal;
    }

    1 reference | Sven, 13 days ago | 1 author, 1 change
    public int Pop()
    {
        if (IsEmpty)
            throw new Exception("The stack is empty. Pop is not allowed");

        return stackArray[top--];
    }
```

```
public class Stack<T>
{
    private T[] stackArray;
    private int top = -1;

    3 references | Sven, 1 day ago | 1 author, 1 change
    public Stack(int InitialSize = 5)
    {
        stackArray = new T[InitialSize];
    }

    6 references | Sven, 1 day ago | 1 author, 1 change
    public void Push(T item)
    {
        if (IsFull)
        {
            var newArray = new T[stackArray.Length * 2];
            System.Array.Copy(stackArray, newArray, stackArray.Length);
            stackArray = newArray;
        }

        stackArray[++top] = item;
    }

    1 reference | Sven, 1 day ago | 1 author, 1 change
    public T Pop()
    {
        if (IsEmpty)
            throw new Exception("The stack is empty. Pop is not allowed");

        return stackArray[top--];
    }
```

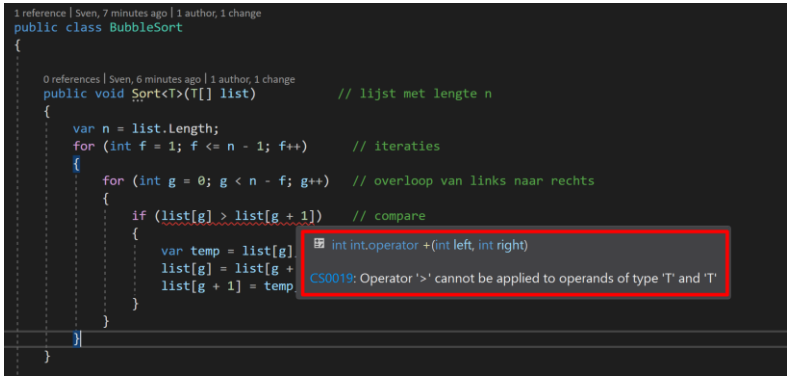
Genric stack gebruiken

```
public class Stack<T>
{
    private T[] stackArray;
    private int top = -1;

    3 references | Sven, 1 day ago | 1 author, 1 change
    public Stack(int InitialSize = 5)
    {
        stackArray = new T[InitialSize];
    }

    6 references | Sven, 1 day ago | 1 author, 1 change
    public void Push(T item)
    {
        if (IsFull)
        {
            var newArray = new T[stackArray.Length * 2];
            System.Array.Copy(stackArray, newArray, stackArray.Length);
            stackArray = newArray;
        }

        stackArray[++top] = item;
    }

    1 reference | Sven, 1 day ago | 1 author, 1 change
    public T Pop()
    {
        if (IsEmpty)
            throw new Exception("The stack is empty. Pop is not allowed");

        return stackArray[top--];
    }
```

```
0 references | 0 changes | 0 authors, 0 changes
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");

    var stack = new Stack<int>(20);
    stack.Push(10);
    stack.Push(12);
    stack.Push(5);

    int last = stack.Pop();
    //.....

    var stack2 = new Stack<double>(10);
    stack2.Push(10.34);
    stack2.Push(22.99);
    //...

    var stack3 = new Stack<Auto>(20);
    stack3.Push(new Auto());
    //...
}
```

Generics: hoe objecten sorteren

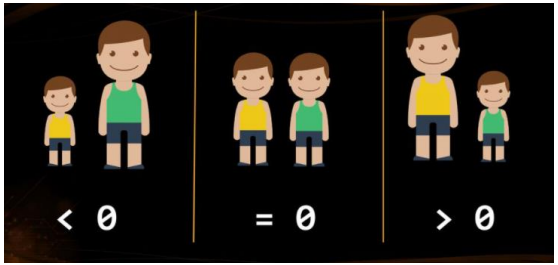
- Voor een **sorteeralgoritme** moeten we 2 elementen met elkaar kunnen vergelijken (de '**compare**' operatie)
- Het .NET framework kan echter niet "out of the box" een compare doen **voor eender wel type**.
  - bv. hoe 2 auto's met elkaar vergelijken? Welke komt eerst in de gesorteerde lijst.
  - Is dat op basis van aantal km, bouwjaar, kleur,… ?



IComparable

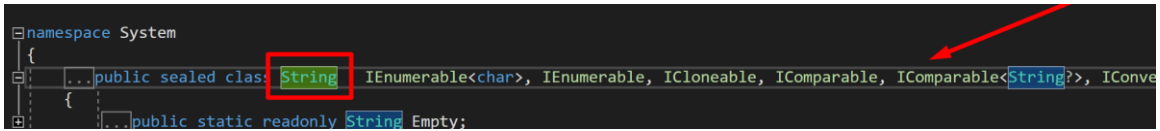
- Er moet een **systeem** bedacht worden waardoor we 2 objecten van **eender welk type** met elkaar kunnen vergelijken.
- Dit bestaat reeds onder de vorm van een Interface : **IComparable**

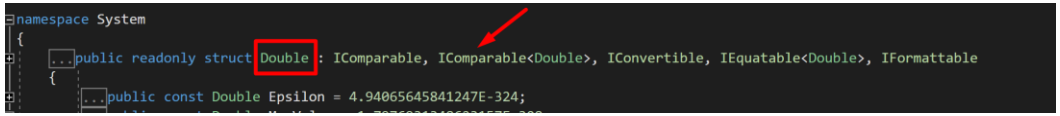
```
namespace System
{
    public interface IComparable<in T>
    {
        //
        // Summary:
        //     Compares the current instance with another object of the same type and returns
        //     an integer that indicates whether the current instance precedes, follows, or
        //     occurs in the same position in the sort order as the other object.
        //
        // Parameters:
        //   other:
        //     An object to compare with this instance.
        //
        // Returns:
        //     A value that indicates the relative order of the objects being compared. The
        //     return value has these meanings:
        //     Value - Meaning
        //     Less than zero - This instance precedes other in the sort order.
        //     Zero - This instance occurs in the same position in the sort order as other.
        //     Greater than zero - This instance follows other in the sort order.
        int CompareTo(T? other);
    }
```



IComparable

- **Alle c# types** (int, string, double, bool..) implementeren reeds deze interface !
- Dit is een **uniforme** manier om 2 elementen van eender welk type met elkaar te kunnen vergelijken.





Generics: restricties opleggen

- We kunnen bij een generic type een restrictie opleggen

```
1 reference | Sven, 19 minutes ago | 1 author, 1 change
public class BubbleSort
{
    0 references | Sven, 25 minutes ago | 1 author, 1 change
    public void Sort<T>(T[] list) where T: IComparable<T>  // lijst met lengte n
    {
        var n = list.Length;
        for (int f = 1; f <= n - 1; f++)      // iteraties
        {
            for (int g = 0; g < n - f; g++)   // overloop van links naar rechts
            {
                if (list[g].CompareTo(list[g + 1]) > 0)    // compare
                {
                    var temp = list[g];       // swap
                    list[g] = list[g + 1];
                    list[g + 1] = temp;
                }
            }
        }
    }
```

restrictie: enkel types die IComparable implementeren kunnen gesorteerd worden met dit algoritme

De compiler "weet" nu dat elk type een methode CompareTo(..) heeft

Ik kan alle .NET types nu sorteren, maar nog niet alle zelfgemaakte klassen

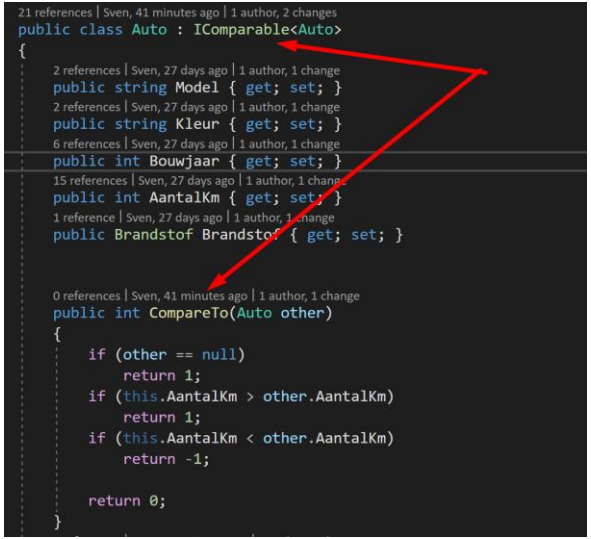
```
var bubbles = new BubbleSort();
bubbles.Sort<Auto>(list);

bubbles.Sort
```

void BubbleSort.Sort<Auto>(Auto[] list) (+ 2 overloads)
Sort the list using the default comparer.

CS0311: The type 'MyLibrary.Auto' cannot be used as type parameter 'T' in the generic type or method 'BubbleSort.Sort<T>(T[])'. There is no implicit reference conversion from 'MyLibrary.Auto' to 'System.IComparable<MyLibrary.Auto>'.

IComparable: wat met onze eigen klassen ?

- Stel: we willen een lijst van ‘Auto’ objecten laten sorteren
- Dan moeten we onze klasse de interface **IComparable** laten implementeren.
- Nu kunnen we ook auto’s laten sorteren.
- De **klasse** ‘Auto’ **bepaalt zelf** hoe de sortering verloopt (in dit voorbeeld **volgens “aantalKm**”)



Sorteren op meerdere manieren ?

- Ik wil graag 2 lijsten van Auto's maken waarbij:
  - De ene lijst is gesorteerd op '**Aantalkm**'
  - De andere lijst is gesorteerd op '**Bouwjaar**'
- Hoe kan ik dit realiseren als de klasse **zelf** de 'Compare' doet (en dus de sortering bepaalt) … ?
- Niet: ik moet **een andere partij** de 'Compare' laten doen.
- Hiervoor bestaat een andere interface: '**IComparer**'

IComparer: de ‘compare’ uitbesteden

- Een ‘Comparer’ klasse doet niet meer dan 2 objecten van hetzelfde type met elkaar vergelijken
- Dit gebeurt adhv. de ‘Compare(..)’ methode.

```
namespace System.Collections.Generic
{
    public interface IComparer<in T>
    {
        //
        // Summary:
        //     Compares two objects and returns a value indicating whether one is less than,
        //     equal to, or greater than the other.
        //
        // Parameters:
        //   x:
        //     The first object to compare.
        //
        //   y:
        //     The second object to compare.
        //
        // Returns:
        //     A signed integer that indicates the relative values of x and y, as shown in the
        //     following table.
        //     Value - Meaning
        //     Less than zero -x is less than y.
        //     Zero -x equals y.
        //     Greater than zero -x is greater than y.
        int Compare(T? x, T? y);
    }
```

Verschillende implementaties

- We kunnen nu verschillende ‘Comparers’ bouwen:

```
public class CarComparerByKm : IComparer<Auto>
{
    1 reference | Sven, 17 hours ago | 1 author, 1 change
    public int Compare(Auto x, Auto y)
    {
        if (x == null || y == null)
        {
            if (x == null && y == null)
                return 0;
            if (x == null)
                return -1;
            return 1;
        }
        if (x.AantalKm < y.AantalKm)
            return -1;
        if (x.AantalKm > y.AantalKm)
            return 1;

        return 0;
    }
}
```

```
class CarComparerByBouwJaar : IComparer<Auto>
{
    – references | - changes | -authors, -changes
    public int Compare(Auto x, Auto y)
    {
        if (x == null || y == null)
        {
            if (x == null && y == null)
                return 0;
            if (x == null)
                return -1;
            return 1;
        }
        if (x.Bouwjaar < y.Bouwjaar)
            return -1;
        if (x.Bouwjaar > y.Bouwjaar)
            return 1;

        return 0;
    }
}
```

Sorteren mbv. een ‘Comparer’

- De bubblesort met een Comparer

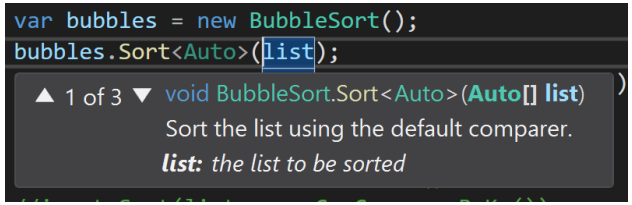
```
/// <summary>
/// Sort the list, using the specified comparer
/// </summary>
/// <typeparam name="T">type of the list elements</typeparam>
/// <param name="list">the list to be sorted</param>
/// <param name="comparer">the comparer which will determine the order</param>
1 reference | 0 changes | 0 authors, 0 changes
public void Sort<T>(T[] list, IComparer<T> comparer)  // lijst met lengte n
{
    var n = list.Length;
    for (int f = 1; f <= n - 1; f++)      // iteraties
    {
        for (int g = 0; g < n - f; g++)   // overloop van links naar rechts
        {
            if (comparer.Compare(list[g], list[g + 1]) > 0)    // compare
            {
                var temp = list[g];       // swap
                list[g] = list[g + 1];
                list[g + 1] = temp;
            }
        }
    }
}
```

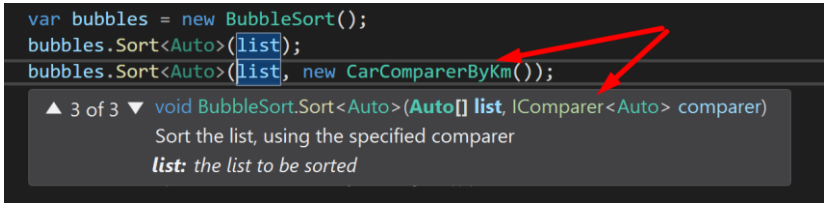
```
//Generic sort with specific Comparer
var list = new Auto[3];
list[0] = new Auto()
{
    AantalKm = 1000
};
list[1] = new Auto()
{
    AantalKm = 20
};
list[2] = new Auto()
{
    AantalKm = 80
};

var bubbles = new BubbleSort();
bubbles.Sort<Auto>(list, new CarComparerByKm());
```

De aanroeper van de sort beslist nu welke ‘comparer’ er gebruikt zal worden

IComparable vs. IComparer

- Gebruik **IComparable** als de ‘default’ comparer
- Gebruik **IComparer** om bijkomende Comparers te bouwen





Wat bestaat er reeds in .NET ?

Generic classes and methods combine reusability, type safety, and efficiency in a way that their non-generic counterparts cannot. Generics are most frequently used with collections and the methods that operate on them. The System.Collections.Generic namespace contains several generic-based collection classes. The non-generic collections, such as ArrayList are not recommended and are maintained for compatibility purposes. For more information, see Generics in .NET.

Of course, you can also create custom generic types and methods to provide your own generalized solutions and design patterns that are type-safe and efficient. The following code example shows a simple generic linked-list class for demonstration purposes. (In most cases, you should use the List<T> class provided by .NET instead of creating your own.) The type parameter `T` is used in several locations where a concrete type would ordinarily be used to indicate the type of the item stored in the list. It is used in the following ways:

Wat bestaat er reeds in .NET ?

- List<T>

The List<T> class is the generic equivalent of the ArrayList class. It implements the IList<T> generic interface by using an array whose size is dynamically increased as required.

Methods such as BinarySearch and Sort use an ordering comparer for the list elements. T

The List<T> is not guaranteed to be sorted. You must sort the List<T> before performing operations (such as BinarySearch) that require the List<T> to be sorted.

In deciding whether to use the List<T> or ArrayList class, both of which have similar functionality, remember that the List<T> class performs better in most cases and is type safe. If a reference type is used for type `T` of the

Wat bestaat er reeds in .NET ?

- LinkedList<T> Represents a doubly linked list.

LinkedList<T> is a general-purpose linked list. It supports enumerators and implements the ICollection interface, consistent with other collection classes in the .NET Framework.

LinkedList<T> provides separate nodes of type LinkedListNode<T>, so insertion and removal are O(1) operations

You can remove nodes and reinsert them, either in the same list or in another list, which results in no additional objects allocated on the heap. Because the list also maintains an internal count, getting the Count property is an O(1) operation.

Each node in a LinkedList<T> object is of the type LinkedListNode<T>. Because the LinkedList<T> is doubly linked, each node points forward to the Next node and backward to the Previous node.

LinkedList<T>.Find(T) Method

Namespace: System.Collections.Generic

Assembly: System.Collections.dll

Finds the first node that contains the specified value.

The LinkedList<T> is searched forward starting at First and ending at Last.

This method performs a linear search; therefore, this method is an O(`n`) operation, where `n` is Count.

Wat bestaat er reeds in .NET ?

- Queue<T> Represents a first-in, first-out collection of objects.

This class implements a generic queue as a circular array. Objects stored in a Queue<T> are inserted at one end and removed from the other. Queues and stacks are useful when you need temporary storage for information; that is, when you might want to discard an element after retrieving its value. Use Queue<T> if you need to access the information in the same order that it is stored in the collection. Use Stack<T> if you need to access the information in reverse order. Use ConcurrentQueue<T> or ConcurrentStack<T> if you need to access the

Three main operations can be performed on a Queue<T> and its elements:

- Enqueue adds an element to the end of the Queue<T>.
- Dequeue removes the oldest element from the start of the Queue<T>.
- Peek peek returns the oldest element that is at the start of the Queue<T> but does not remove it from the Queue<T>.

The capacity of a Queue<T> is the number of elements the Queue<T> can hold. As elements are added to a Queue<T>, the capacity is automatically increased as required by reallocating the internal array. The capacity can be decreased by calling TrimExcess.

Wat bestaat er reeds in .NET ?

- Stack<T>

Represents a variable size last-in-first-out (LIFO) collection of instances of the same specified type.

Stack<T> is implemented as an array.

Three main operations can be performed on a System.Collections.Generic.Stack<T> and its elements:

- Push inserts an element at the top of the Stack.
- Pop removes an element from the top of the Stack<T>.
- Peek returns an element that is at the top of the Stack<T> but does not remove it from the Stack<T>.

The capacity of a Stack<T> is the number of elements the Stack<T> can hold. As elements are added to a Stack<T>, the capacity is automatically increased as required by reallocating the internal array. The capacity can be decreased by calling TrimExcess.

If Count is less than the capacity of the stack, Push is an O(1) operation. If the capacity needs to be increased to accommodate the new element, Push becomes an O(`n`) operation, where `n` is Count. Pop is an O(1) operation.

Wat bestaat er reeds in .NET ?

- SortedSet<T> — Represents a collection of objects that is maintained in sorted order.

A SortedSet<T> object maintains a sorted order without affecting performance as elements are inserted and deleted. Duplicate elements are not allowed. Changing the sort values of existing items is not supported and may lead to unexpected behavior.

Constructors

SortedSet<T>()	Initializes a new instance of the SortedSet<T> class.
SortedSet<T>(IComparer<T>)	Initializes a new instance of the SortedSet<T> class that uses a specified comparer.

Wat bestaat er reeds in .NET ?

- Dictionary<Tkey,TValue>

Represents a collection of keys and values.

The Dictionary<TKey,TValue> generic class provides a mapping from a set of keys to a set of values. Each addition to the dictionary consists of a value and its associated key. Retrieving a value by using its key is very fast, close to O(1), because the Dictionary<TKey,TValue> class is implemented as a hash table.

> **Note**
>
> The speed of retrieval depends on the quality of the hashing algorithm of the type specified for `TKey`.

The capacity of a Dictionary<TKey,TValue> is the number of elements the Dictionary<TKey,TValue> can hold. As elements are added to a Dictionary<TKey,TValue>, the capacity is automatically increased as required by reallocating the internal array.

> **Important**
>
> We don't recommend that you use the `Hashtable` class for new development. Instead, we recommend that you use the generic **Dictionary<TKey,TValue>** class. For more information, see **Non-generic collections shouldn't be used** on GitHub.

Wat bestaat er reeds in .NET ?

- SortedDictionary<Tkey, Tvalue>

Represents a collection of key/value pairs that are sorted on the key.

The SortedDictionary<TKey,TValue> generic class is a binary search tree with O(log n) retrieval, where n is the number of elements in the dictionary. In this respect, it is similar to the SortedList<TKey,TValue> generic class.

SortedDictionary<TKey,TValue> requires a comparer implementation to perform key comparisons. You can specify an implementation of the IComparer<T> generic interface by using a constructor that accepts a `comparer` parameter; if you do not specify an implementation, the default genericcomparer Comparer<T>.Default is used. If type `TKey` implements the System.IComparable<T> generic interface, the default comparer uses that implementation.

- SortedList<TKey,TValue> uses less memory than SortedDictionary<TKey,TValue>.
- SortedDictionary<TKey,TValue> has faster insertion and removal operations for unsorted data: O(log n) as opposed to O(n) for SortedList<TKey,TValue>.
- If the list is populated all at once from sorted data, SortedList<TKey,TValue> is faster than SortedDictionary<TKey,TValue>.


That's all Folks!