Week 3

Last updated by | kees.vanloenen | Jul 3, 2025 at 4:12 PM GMT+2

0630 Dag 9/12

- Daily Scrum - vragen aan elkaar
- Code Review
- Afspraken mbt. hoofd- en kleine letters:

```
private const double MaxSpeed = 120.05;     // PascalCase
internal record Person {};                  // PascalCase
private int _count;                         // CamelCase + underscore
public int Count { get; private set; }      // PascalCase
private void DoeIets(int startValue);       // methode PascalCase, params CamelCase
public enum Kleur {}                        // Pascal Case en enkelvoud
public interface IComparable {}             // Pascal Case en start met I
```

- Ondersteuning voor foreach vereist dat de IntList een `GetEnumerator()` methode implementeert. Implementeer daarvoor de `IEnumerable` interface.
  Je zou nu ambachtelijk een iterator kunnen schrijven. Gebruik liever de `yield return` syntax...

Bij de niet-generieke `IntList` ziet dit er als volgt uit:

```
public IEnumerator GetEnumerator()
{
    // Net als in Harry Potter ...
    // De compiler maakt hier onderwater weer een hele enumerator van:

    for (int index = 0; index < _count; index++)
    {
        yield return _items[index];
    }
}
```

Later bij een generieke `MyList<T>` ziet dit er als volgt uit:

```
IEnumerator IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}

public IEnumerator<int> GetEnumerator()
{
    for (int index = 0; index < _count; index++)
    {
        yield return _items[i];
    }
}
```

- IntList generiek gemaakt: op sommige plekken `int` vervangen door `T`:

```
public class MyList<T> : IEnumerable
{
    private T[] _items;
    // ...
}
```

(PS liever `IEnumerable<T>` natuurlijk)

- Generics, je wilt een class of bijv. methode ondersteuning laten bieden voor verschillende data typen:

  ipv. `int` gebruiken we bij de definitie een `T`
  `<T>` betekent hier ik geef een T als type mee

```
private static void PrintEersteElement<T>(T[] waarden)
{
    Console.WriteLine(waarden[0]);
}

// aanroeper:
static void Main(string[] args)
{
    int[] getallen = { 1,2,3 };
    PrintEersteElement<int>(getallen);
    //                 ^^^^^ meestal niet nodig
}
```

- Generic constraints:

```
static void PrintSmallest<T>(List<T> items) where T : IComparable<T>
```

  Het type dat de aanroeper meegeeft MOET een `IComparable` interface implementeren
  Ik weet dus zeker dat het type dat ik meegeef een `CompareTo` methode aan boord heeft.

- Delegates intro:
  Om code duplicatie te voorkomen is het soms handig dat je een methode als parameter meegeeft aan een andere methode...
  Omdat alles getypeerd is in C#, geldt dat ook voor een methode. Het type voor een methode wordt een `delegate` genoemd. Je definieert hem in of buiten een class:

```
//                    vvvvvvvvvvvvv   nieuw type!
public delegate double MathsFunction(double arg);
//              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^   zo moet de functie eruit zien
```

  (vergelijk het met het koffertje waar precies 1 type gun in past)

en gebruikt hem vervolgens bijv als *parameter type* in:

```
static void Plot(MathsFunction f, double startX, double endX, double step)
//                  👆
{
    for (double x = startX; x <= endX; x += step)
    {
        double y = f(x);   // hier roepen we aan wat in f is meegegeven
        Console.WriteLine($"{x:N2}, {y:N2}");
    }
}
```

PS. delegates kun je ook gebruiken bij variabelen, fields, return type

- De `Plot` methode kunnen we aanroepen met allerlei soorten functions zolang de signatuur maar overeenkomt met die van de delegate (double in, double uit):

```
plot(Math.Tan, ...);
plot(Dubbelaar, ...);
plot(x => x * 3, ...);
```

- git:
  - 3 hoofdlocaties: working directory | index | object database
  - status van een file: untracked | unmodified | modified | staged
- git uitvoeren van de eerste 3 labs

0701 Dag 10/12

- Daily Scrum
- Code Review
- Doorgegaan met Plot voorbeeld van gisteren, de delegate is een type waar een methode met een specifieke signatuur in past:

  ```
  public delegate double MathsFunctie(double arg);
  ```

- Plot methode kan worden aangeroepen met:

  ```
  Plot(f: Math.Sin, xStart: 0.0, xEnd: Math.PI / 2, step: 0.01);
  Plot(Math.Cos, 0.0, Math.PI / 2, 0.01);
  Plot(new MathsFunctie(Math.Cos), 0.0, Math.PI / 2, 0.01);   // C# onder water
  Plot(mijnFunctie, 0.0, Math.PI / 2, 0.01);
  Plot(Trippelaar, 0.0, Math.PI / 2, 0.01);
  Plot(x => x * 3.5, 0.0, Math.PI / 2, 0.01);
  ```

- Als oplossing voor de delegate heeft Microsoft enkele generic delegates uitgevonden:
  - `Func` retourneert een waarde
  - `Action` retourneert geen waarde
  - `EventHandler` retourneert geen waarde en wordt specifiek bij events gebruikt
- We hadden:

  ```
  static void Plot(MathsFunctie f, double xStart, double xEnd, double step)
  ```

  en dat wordt nu:

  ```
  static void Plot(Func<double, double> f, double xStart, double xEnd, double step)
  ```

  De expliciet uitgeschreven delegate is nu niet meer nodig!
- Het laatst argument van een Func is altijd zijn return waarde:

  ```
  Func<string, int, double>
  ```

  De methode die 'hierin past' heeft 2 input parameters: `string` en `int`, hij retourneert een `double`.
- Events:
  Stap voor stap een event invoker en event handler geschreven en elke stap uitgelegd.
  Ook het Microsoft event pattern toegepast, meer info:
  https://dev.azure.com/kc-academy/DemosJuni2025/_wiki/wikis/DemosJuni2025.wiki/60/MS-Event-Pattern

- In plaats van een expliciete delegate mag je hier ook weer een ingebouwde delegate gebruiken:

```
public event EventHandler<VijandGespotArgs>? VijandGespot;
```

- In testen kun je je subscriben op een event om te testen of het geraised wordt:

```
// Arrange
bool eventRaised = false;
_sut.VijandGespot += (s, e) => eventRaised = true;

// Act
_sut.SpotVijand("Sector 7");

// Assert
Assert.IsTrue(eventRaised);
```

- git: local branch, remote tracking branch en remote branch

0703 Dag 11/12

- Grote herhalingsquiz met uitleg
- Code Review
- Uitwerking van SortedIntList (TDD)
- Uitwerking van Generics
- OO opdracht, putting it together, eerst individueel daarna in duo's
- Tuples:
  - tot C# 7.0 reference types, daarna ook value types:
  - deconstructen:

```
var t2 = (id: 100, naam: "Bo", samenwonend: false);
Console.WriteLine(t2.naam);
// (int id, string naam, bool samenwonend) = t2;
var (id, naam, samenwonend) = t2;
```

    Handig om waardes te swappen:

```
int a = 5;
int b = 10;

// swap met hulp variabele
// int temp = a;
// a = b;
// b = temp;

// swap met tuple
(a, b) = (b, a);
```

  - een methode kan meer waarden retourneren in een tuple:

```
(int som, int verschil) Bereken(int x, int y)
{
    return (x + y, x - y);
}
```

- herhaling `delegate`, `Func<int, int>` en lambdas:

```
Func<int, int> kwadraat = x => x * x;
```

- `FindAll()` behandeld (geen echte LINQ extension method, maar lijkt er wel op):

```
static bool IsLarge(int n)
{
    return n > 18;
}


// FindAll() verwacht een predicate (1 input parameter, boolean output)
// Func<int, bool> predikaat = IsLarge;
Predicate<int> predikaat = IsLarge;

List<int> grotePriemGetallen = priemgetallen.FindAll(predikaat);
```

- Anonymous types, veel gebruikt in LINQ ('projecting' - welke combinatie van velden wil ik bijv. ophalen):

```
var ambtenaar1 = new { Naam = "Mo", Plaats = "Ede" };
var ambtenaar2 = new { Naam = "Ad", Plaats = "Urk" };
```

  Onderwater maakt C# hiervoor een class aan

- LINQ kent 2 soorten syntax: Query en Method:

```
// Optie 1: Query syntax
var urkers = from burger in burgers
             where burger.Plaats == "Urk"
             select burger;

// Optie 2: Method syntax
var urkers2 = burgers.Where(burger => burger.Plaats == "Urk");
```

- Lazy execution: de query wordt pas uitgevoerd bij een zgn. "execution enforcing" statement: .ToList(), .ToArray(), .Count, foreach loop

- Willen we een combinatie van velden ophalen, gebruik dan achter de `select` een instantie van een class of record, of een anonymous type (of een tuple)