Week 1

Last updated by | kees.vanloenen | Jun 19, 2025 at 4:08 PM GMT+2

Week 1

0616 Dag 1/12

- Kennismaken:
  - Tekenwedstrijd
  - Wapenschilden
  - Shu Ha Ri
  - Huisregels
  - Voorkennisvisualisatie
  - Omgaan met niveauverschillen
- Solution bestaat uit tenminste 1 project maar kan uit meerdere projecten bestaan
  (1 van die projecten is het startup project)
- Data types zijn in C# óf **value types** óf **reference types**, vanmiddag focus op **value types**:

Type	Grootte (bytes)	Bereik	Categorie
byte	1	0 tot 255	Integraal
(sbyte)	1	-128 tot 127	Integraal
short	2	-32.768 tot 32.767	Integraal
(ushort)	2	0 tot 65.535	Integraal
int	4	-2.147.483.648 tot 2.147.483.647	Integraal
(uint)	4	0 tot 4.294.967.295	Integraal
long	8	-9.223.372.036.854.775.808L tot 9.223.372.036.854.775.807**L**	Integraal
(ulong)	8	enkel positief **UL**	Integraal
float	4	groot F	Decimaal niet precies
double	8	heel groot **D**	Decimaal niet precies
decimal	16	heel heel groot **M**	Decimaal precies
char	2	Unicode character bijv. 'a'	Overig
bool	1	`true` of `false`	Overig

(data types tussen haakjes hoef je niet uit het hoofd te weten)

- de namen van de data types zijn C# aliases voor data types in het **Common Type System (CTS)**. Bijv:
  - `int` in C# is een alias voor `Int32` in CTS
  - `float` in C# is een alias voor `Single` in CTS
    Het CTS is op haar beurt onderdeel van de **Common Language Runtime (CLR)**. Dit is het onderdeel van het .NET-platform dat verantwoordelijk is voor het uitvoeren van .NET-programma's. De CLR beheert onder andere geheugen, voert garbage collection uit, zorgt voor zaken als type safety en exception handling. Met de CLR is het mogelijk om code geschreven in verschillende .NET-talen (zoals C#, C++, VB.NET en F#) op dezelfde manier uit te voeren.
- gebruik een `decimal` voor geldbedragen en percentages (financiële applicaties)
- Concateneren kan op drie manieren:
  - handmatig: `Console.WriteLine("Je bent " + leeftijd + " jaar");` (snel)
  - string interpolatie: `Console.WriteLine($"Je bent {leeftijd} jaar");` (leesbaar)
  - composite formatting: `Console.WriteLine("Je bent {0} jaar", leeftijd);` (wat ouder)
- Bij string interpolatie, gebruik optioneel `:` voor opmaak

```
decimal prijs = 12345.6789m;
$"Het kost {prijs}!"     // 123456,6789
$"Het kost {prijs:F3}!"  // 123456,679
$"Het kost {prijs:C}!"   // € 12.345,6 (afhankelijk van culture)
$"Het kost {prijs:N3}!"  // 12.345,679
```

  - tip: `Console.OutputEncoding = Encoding.UTF8; // toon ook bepaalde karakters als €`
- `Console.ReadLine()` verwacht input van de gebruiker, afgesloten met ENTER
  - string leeftijd = Console.ReadLine();
  - typt de gebruiker enkel ENTER dan wordt leeftijd ""
  - typt de gebruiker 27 gevolgd door ENTER dan wordt leeftijd "27"
  - typt de gebruiker CTRL+Z gevolgd door ENTER dan wordt leeftijd `null`
- string leeftijd = Console.ReadLine() ?? ""; // retourneert ReadLine() null, maak er "" van
- ternary operator: expressie ? indien waar : indien onwaar;
  string groeiFase = leeftijd >= 18 ? "volwassen" : "kind";
- `int.Parse(LeeftijdAlsString)` converteert een `string` naar een `int`
- `int.TryParse(LeeftijdAlsString, out int leeftijd)` probeert dat ook, lukt het dan wordt `true` geretourneerd, anders `false`. De `out` variabele `leeftijd` bevat als het gelukt is de gewenste integer.
- met `ref` of `out` geven we het **adres** van een variabele mee. Er zijn enkele verschillen:
  - `ref` => de variabele MOET geïnitialiseerd zijn en de methode MAG hem overschrijven
  - `out` => de variabele MAG geïnitialiseerd zijn en de methode MOET hem vullen
- `Math.Round` rond standaard af volgens het **"to even"**-principe. Dit betekent dat bij een getal precies halverwege (zolas 2.5) wordt afgerond naar het dichtstbijzijnde **even** getal:

```
Math.Round(2.5);      // 2
Math.Round(3.5);      // 4
Math.Round(2.455, 2); // 2.46
Math.Round(2.445, 2); // 2.44
```

Wil je altijd **omhoog** afronden bij .5, dan kun je de overload met een `MidpointRounding`-parameter gebruiken:

```
Math.Round(2.5, MidpointRounding.AwayFromZero); // 3
Math.Round(3.5, MidpointRounding.AwayFromZero); // 4
Math.Round(2.455, 2, MidpointRounding.AwayFromZero); // 2.46
Math.Round(2.445, 2, MidpointRounding.AwayFromZero); // 2.45
```

0617 Dag 2/12

- Daily Scrum
- Code Review
- (Git) 1. working dir | 2. index (staging area) | 3. object database
  - `git add .`
  - `git commit -m "completed bugfix 123"`
  - `git push`
- type casting: impliciet en expliciet
  - impliciet = automatisch (geen gegevensverlies)

```
byte greenCode = 253;
int color = greenCode;
```

  - expliciet = jijzelf gebruikt daarvoor haakjes (gegevensverlies)

```
double doubleWaarde = 9.78d;
int intWaarde = (int)doubleWaarde;
```

- Waarom liever `TryParse()` ipv `Convert.ToInt32()` bij string conversie?
  - `Convert. ...` vereist exception handling:
    `try-catch-finally`
- eenvoudige enum
- `if`, `else if`, `else`
- ternary operator: expressie ? indien waar : indien onwaar;
- switch statement (inclusief fall through en `or` pattern matching)
- switch expression (inclusief gebruik `or` en de `_` discard)
- Lussen: `for` en `foreach`
- StringBuilder vs handmatig concateneren
- Methods en parameters
  - optionele parameters (default values)
  - `params` array

0618 Dag 3/12

- Daily Scrum
- Code Review
- Debuggen met IDE:
  - breakpoints(F9)
  - step into(F11)
  - step over(F10)
  - step out(SHIFT+F11)
  - immediate window (vars in huidige regel + regel erboven)
  - locals window (vars in huidige scope)
  - watch window
  - call stack
  - conditionele breakpoints
- `while` vs `do while` (laatste wordt tenminste 1x uitgevoerd)
- Method overloading
- Arrays:
  - eendimensionale arrays, bijv `int[5]`
  - multidimensionale arrays, rectangular: `int[3,4]`
- |Jouw code| *build* ➡️ |Intermediate Language (IL)| ➡️ *just-in-time compile (JIT)* ➡️ |Machine code|
- IL Spy 👈 bekijk IL code (en ontdek bijv. dat verschillende manieren om een array te declareren/initialiseren leiden tot exact dezelfde code)
- index en range operator:

```
int[] arr = { 1, 2, 3, 4, 5 };
var eersteDrie = arr[..3];      // {1, 2, 3}
var laatsteTwee = arr[^2..];    // {4, 5}
var midden = arr[1..4];         // {2, 3, 4}
var zonderLaatste = arr[..^1];  // {1, 2, 3, 4}
```

- samen met Luuk: destructuren:

```
var arr = ['a', 'b', 'c'];
var arr2 = [..arr];  // ECHTE kopie, 2de array in de managed heap
var arr3 = [..arr, 'd', 'e'];
```

- unit testen:
  - MSTest-project, referentie naar library, sut = system under test, `[TestClass]` en `[TestMethod]` zijn *data attributes* (aka annotations)
  - Test-Driven Development (TDD):
    1. schrijf test
    2. run en zie dat hij faalt
    3. schrijf implementatie code
    4. run en zie dat hij slaagt
    5. refactor
  - Probeer zo klein mogelijk te beginnen, stukje bij beetje bouw je de logica
- string s1 = "a"
  string s2 = "a"
  s1 == s2 retourneert true, ondanks dat het reference types zijn, dit komt omdat de == operator overloaded is

---

0619 Dag 4/12

- Daily Scrum
- Code Review
- `git log`, `git log --oneline`
- Procedureel vs OO programmeren (Paul & Olivia)
- OO: Abstractie / Encapsulatie / Polymorfisme / Overerving
- Classes vs Objects
- Fields vs Properties:
  - auto-implemented (tip: type "prop" in class)
  - uitgeschreven property met backing field
- Uitgeschreven property biedt mogelijkheid tot extra logica in getter en/of setter:

  (accolades weggelaten ivm ruimte)

```
private string _naam;   // backing field

public string Naam
{
    get
    {
        // Geef de naam altijd in hoofdletters terug
        return naam.ToUpper();
    }
    set
    {
        naam = value;
    }
}
```

```
private int _leeftijd;  // backing field

public int Leeftijd
{
    get { return leeftijd; }

    set
    {
        if (value < 0)
            leeftijd = 0;  // Negatieve waarden worden 0
        else
            leeftijd = value;
    }
}
```

- Methods (voor gedrag)
- Object initialiser syntax
  - `var kazerne1 = new Kazerne() { Id = 10, Plaats = "Urk" };`
- Access Modifiers: private vs public
- Constructor
  - tip: type "ctor" in class
  - `var kazerne1 = new Kazerne(10, "Urk);`
- Wie mag een property instellen als:
  - `public ... { get; }` 👉 constructor
  - `public ... { get; private set; }` 👉 constructor, methodes in de class
  - `public ... { get; set; }` 👉 constructor, alle methodes
  - `public ... { get; init; }` 👉 constructor, object initializer
- `!` = null-forgiving operator
  ik beloof dat deze waarde nooit null zal worden
- Wat is een assembly: .exe of .dll
- Constructor Overloading:

```
public class Drone
{
  public string Kenteken { get; }
  public Kleur Kleur { get; private set; }

  public Drone(string kenteken):this(kenteken, Kleur.LegerGroen)
  {
  }

  public Drone(string kenteken, Kleur kleur)
  {
    Kenteken = kenteken;
    Kleur = kleur;
  }
}
```

- Unit testen:
  - gebruik een met `[TestInitialize]` gedecoreerde methode voor het instantiëren van objecten die in iedere test nodig zijn:

```
private BmiCalculator _sut = null!;

[TestInitialize]
public void Initialize()
{
    // Arrange
    _sut = new BmiCalculator();
}
```