Week 2

Last updated by | kees.vanloenen | Jun 26, 2025 at 4:08 PM GMT+2

0623 Dag 5/12

- Daily Scrum
- Code Review
- Value Types vs Reference Types presentatie
- static members:

```
private static long _nextId = 1000;     // static = leeft op class, niét instantie
public long Id { get; }                 // iedereen mag lezen

public Horloge(double gewicht)          // static member wordt bij elke initialisatie opgehoogd met 1
{
    Id = _nextId++;
}
```

- `public static class Utils { ... }`
  - alle members van een static class MOETEN static zijn,
  - in een niet static class MOGEN members static zijn
- `try` , `catch` , `finally` en `using`

```
try
{
    /* mogelijk exception veroorzakende code */
}
catch(DivideByZeroException ex)
{
    /* gebruik ex (ex.Message, ex.ParamName) */
}
catch(IndexOutOfRangeException ex)
{
    /* gebruik ex (ex.Message, ex.ParamName) */
}
catch (Exception ex)
{
    /* generieke exception handler als fallback */
}
finally
{
    /* deze code wordt altijd uitgevoerd */
}
```

- `throw new ArgumentException("melding", nameof(Weight));`
  `nameof(Weight)` is beter dan `"Weight"` , omdat het meeverandert als je de property `Weight` later wijzigt in iets anders
- `using` -statement zorgt ervoor dat het object (een `StreamReader`) automatisch wordt opgeruimd zodra de code binnen het using-blok is uitgevoerd. Dit gebeurt door de `Dispose` -methode van het object aan te roepen:

```
string filePath = "mvd.txt";

// Using statement zorgt ervoor dat StreamReader automatisch wordt gesloten
using (StreamReader reader = new StreamReader(filePath))
{
    string content = reader.ReadToEnd();
    Console.WriteLine(content);
}
```

dit wordt onder water:

```
string filePath = "mvd.txt";

StreamReader reader = new StreamReader(filePath);
try
{
    string content = reader.ReadToEnd();
    Console.WriteLine(content);
}
finally
{
    reader.Dispose();
}
```

- `enum`

```
public enum Weer
{
    Zonnig,
    Regenachtig,
    Winderig,
    Sneeuw,
}
```

  - doel: een verzameling van gerelateerde constante waarden met een beschrijvende naam (in enkelvoud)
  - starten met ander cijfer dan 0 (bijv. `Zonnig = 1,`)
  - ander data type bijv `public enum Weer: short`
  - gebruik van `[Flags]` (zie voorbeeld)
- Extension methods:
  - doel: nieuwe methodes aan bestaande types toevoegen (class, interface, enum) zonder deze zelf te wijzigen
  - hoe: 1. static class met 2. static method, 3. eerste argument start met `this`
  - best practise: enkel bij interface of enum
  - heb je een `enum Weer` zet dan op hetzelfde blad: `public static class WeerExtensions` waarbinnen je de extension methods definieert
- Boxing: (value type -> object)
  - veel methodes verwachten object of object[]
  - dan moet je een value type mogelijk boxen (hier hoef je niets voor te doen, het gebeurt impliciet)
  - unboxen gebeurt expliciet, dmv casten: `int resultaat = (int)myObj;`
- Casting:

```
// handmatig
if (item.GetType() == typeof(int))
    totaal += (int)item;
```

```
// handmatig met `as`
int? nummer = item as int?;

if (nummer != null)
    totaal += nummer;
```

```
// pattern matching met `is`
if (item is int)
    totaal += (int)item;
```

```
// nogmaals, maar nu met handig declaration pattern
if (item is int nummer)
    totaal += nummer;
```

0624 Dag 6/12

- Herhalingsquiz
- Korte code review
- git branch en git merge demo:

```
pwd
ls
cd ..
cd foldernaam

git status
git log
git log --oneline
git log --oneline --graph --all --decorate
git config --global alias.mijnhandigenaam "log --oneline --graph --all --decorate"
git branch                # welke branches zijn er (* = actieve branch)
git switch branchnaam     # ga naar branch
git switch -c branchnaam  # maak én ga naar branch
git merge                 # roep je aan vanuit de branch WAARNAARTOE je wilt mergen
git branch -d branchnaam  # delete een branchnaam die je niet langer nodig hebt omdat je hem bijv. heb
```

- Een branch en HEAD zijn in feite labeltjes:
  branch = verwijst naar laatste commit op een 'tak'
  HEAD = verwijst naar de laatste commit op de 'tak' waar je nu bent
- Vaak hoef je bij een merge niets te doen, dit heet een 'fast forward', moet je conflicten oplossen (master en de feature branch hebben conflicterende wijzigingen), los dan eerst de conflicten op en doe daarna een add + commit
- Gezien: extension methods op een 3th-party class en daarna extension methods op interface
- Kleurtjes in de console:

```
var defaultColor = Console.ForegroundColor;
Console.ForegroundColor = ConsoleColor.Yellow;
```

- Extension methods gebruik je:
  - als je dingen wil extenden waar je niet de eigenaar van bent (zoals de `SuperDuperLogger`)
  - als je een werkende class op productie hebt en er logica bij moet komen. Maak een afweging of je de werkende class gaat aanpassen met het risico van bugs of voeg je een extension method toe
  - vooral op enums (om gedrag toe te voegen) en interfaces
- struct:
  - doel: een lichtgewicht value type dat meerdere gerelateerde variabelen kan groeperen
  - class-alike, MAAR leeft op dus de stack en ondersteunt géén inheritance
  - je wil een struct altijd immutable maken (dat wil zeggen dat eenmaal gevulde data members geen andere waarden meer krijgen)
- immutability in onze struct Time:
  - onze properties Hours en Minutes zijn getters (read-only)
  - een methode zoals `AddMinutes(int minutes)` retourneert een `new`e instantie van Time
- plus-operator overloaden om bijv. twee `Time`s bij elkaar op te tellen:

```
public static Time operator+ (Time l, Time r)
{
    return new Time(l.Hours + r.Hours, l.Minutes + r.Minutes);
}
```

- inheritance:
  - enkel bij een IS-relatie!
  - `public class Trompet: MuziekInstrument {}`
  - `public abstract class Voertuig {} // abstract = Voertuig mag NIET geïnstantieerd worden`
  - `public sealed class Voertuig {} // sealed = Van Voertuig mag NIET worden geërfd`
  - `virtual` method in parent class mag worden `override`n in de child class
- Type Substitutability:
  Onderstaand mag:

```
MuziekInstrument klarinet1 = new Klarinet() { Materiaal = Materiaal.Hout };
```

  Een instantie van een *Klarinet* past in een variabele van het type *MuziekInstrument*; andersom mag niet. Waarom is dat handig?
  1. MuziekInstrument kan als parameter type worden gebruikt, zodat alle instrumenten die van MuziekInstrument overerven meegegeven kunnen worden:

```
private static void RepareerInstrument(MuziekInstrument instrument)
{
    Console.WriteLine($"{instrument.GetType().Name} reparatie ({instrument.Materiaal})");
}
```

  2. Een List van type MuziekInstrument kan worden gevuld met allerlei instrumenten. Daarna kan op elk instrument `Bespelen()` worden aangeroepen:

```
List<MuziekInstrument> orkest = new()
{
    new Klarinet() { Materiaal = Materiaal.Hout },
    new Klarinet() { Materiaal = Materiaal.Hout },
    new Trompet() { Materiaal = Materiaal.Koper },
    new Drums() { Materiaal = Materiaal.Darm },
};

foreach(MuziekInstrument instrument in orkest)
{
    instrument.Bespelen();
}
```

- (edge case) wil je een specifieke methode aanroepen die niet in MuziekInstrument zit, maar wel in bijv. Trompet, dan kun je downcasten:

```
if (instrument is Trompet)
{
    ((Trompet)instrument).Dempen();  // Downcast
}
```

of

```
if (instrument is Trompet trompet)
{
    trompet.Dempen();
}
```

0625 Dag 7/12

- Jesse Special: `?` vs `!` vs `required`
- **`?`**
  - mocht al heel lang bij value type (`int? getal = null`)
  - tegenwoordig ook bij reference type ("nullable reference types"), voordat je een methode of property op zo'n nullable reference type aanspreekt, moet je checken of hij `null` is (dat is fijn, want dan heb je minder `NullReferenceException`s)
- **`!`**
  - `private Auto _sut = null!; // ik beloof dat Auto nooit null zal worden`
  - als jij het beter weet dan de compiler, de compiler ziet bijv. niet dat jij een methode hebt gemaakt die bij initialisatie altijd wordt aangeroepen en een property een waarde toekent
- **`required`**
  - bij een property, bij een object initalisatie moet de developer die een object maakt een waarde voor dit property meegeven
- Inheritance:
  - `virtual` methode in base class mag in child class worden overschreven met `override`
  - `new` in de child class, hides een methode in de base class, polymorfisme wordt nek omgedraaid
  - abstract (method/class)
  - sealed (class)
  - protected (enkel zichtbaar in class zelf en children van de class)
- Interface: altijd public, alle methodes zijn 'abstract' en moeten bij een class die de interface implementeert, een implementatie hebben
- IList<T> erft van ICollection<T> erft van IEnumerable<T>:
  - IEnumerable<T> object kan worden geïtereerd met een `foreach` loop

    - ICollection<T> plus `Count`, `Add()`, `Remove()`, `Clear()`, `Contains()`, ..
    - IList<T> plus `IndexOf()`, `RemoveAt()`, `this[]`, ..
- Record: ingebouwde ToString(), equality is overwritten:

```
record Person(string Naam);
```

```
var p1 = new Person("Ab");
var p2 = new Person("Ab");
p1 == p2;   // true
```

0626 Dag 8/12

- Record:
    - soort databakje
    - we houden objecten het liefst zo immutable mogelijk:
        - belangrijk bij multithreading
        - betere voorspelbaarheid, minder bugs
        - veiliger te delen met andere delen van applicatie (worden nl. niet aangepast)
        - oudere versies blijven behouden (eenvoudig teruggaan naar 'vorige staat' zoals in git)
        - caching (goed te cachen omdat hun waarde nooit verandert)
    - standaard enkel Getter
    - `with` bijv. `Burger burger2 = burger1 with { AantalKinderen = 1 }`
    - inheritance (enkel records onderling)
    - pattern matching in parent is niet echt SOLID (open/closed), maar in sommige scenario's erg leesbaar (business logica bij elkaar):

```
public abstract record Vorm
{
    public double Diameter() => this switch
    {
        ...
        Bol(double r) => 2 * r,
        ...
     };
 }
```

- presentatie ref parameters:
    - value types doorgeven
    - value types by ref doorgeven
    - reference types doorgeven
    - reference types by ref doorgeven <- vaak handig om code te versnellen (zien we straks)
- LINQ = Language InteGrated Query
- Eén manier om met in-memory objecten, databases of xml te communiceren
- evenGetallen is van het type `IEnumerable<int>` in:

```
var evenGetallen = getallen.Where(n => n % 2 == 0);
```

- enkel een execution-enforcing statement zal bovenstaande LINQ uitvoeren:
  - `ToArray()`
  - `ToList()`
  - foreach loop
  - `Count`
- bouwen van een eigen IntList, geheel TDD:
  - array is de basis
  - `Count` property
  - `Add()` method
  - `Resize()` method (volgens implementatie Microsoft)
  - indexer toegevoegd (later validatie toegevoegd):

```
public int this[int index]
{
   get { return _items[index]; }
   set { _items[index] = value; }
}
```