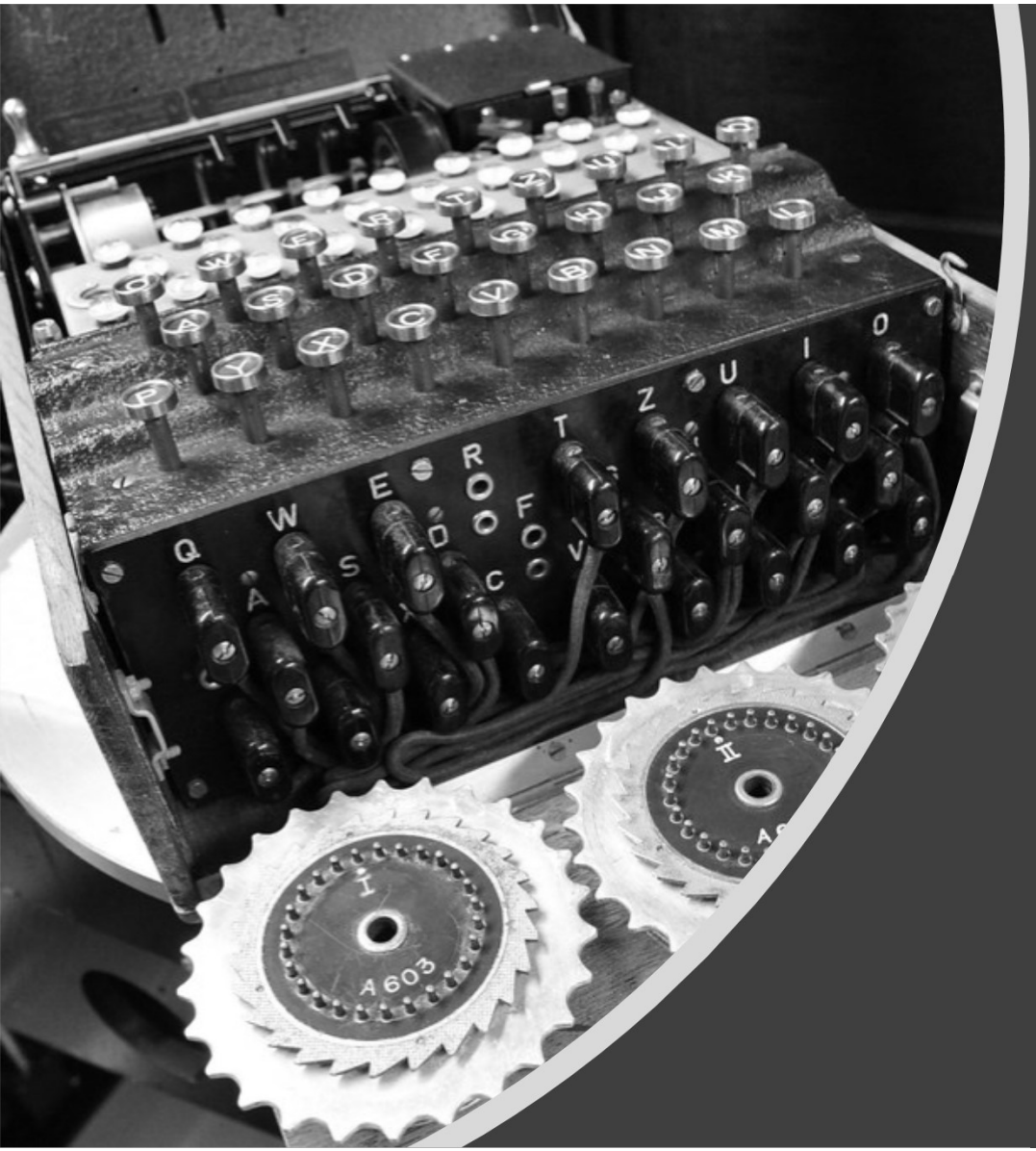


Coderen in C#

Parameters

- Een functie kan géén, één of meerdere parameters declareren.
- Parameters worden standaard doorgegeven als **value.**
  - Veranderingen gemaakt binnen de functie worden niet teruggegeven.
- Parameter modifiers: **out** en **ref**
  - Moeten steeds herhaald worden bij de aanroep!
  - ref:
    - Parameters worden doorgegeven als reference. Veranderingen binnen de functie reflecteren op de parameter.
  - out:
    - Werkt zoals een ref maar moet niet op voorhand toegekend worden.
    - De functie MOET hem toekennen voor de functie beëindigd wordt.

Parameters

- Optionele parameters
  - Het is mogelijk om default waarden aan parameters toe te kennen.
  - De waarde van een optionele parameter moet een constante zijn.
  - Verplichte parameters moeten **voor** optionele parameters geplaatst worden

```
public void OptMethod(int a,string optioneel="default") {…}
```

- Params modifier

```
long res=Multiply(2,4,10,5,612);
```

  - Als laatste parameter kan een ‘params’ parameter geplaatst worden.
  - Een ‘params’ parameter is altijd een array

```
public long Multiply(params int[] nums) {
  long res = 0;
  foreach(int x in nums ?? new int[0]) { res *= x; }
  return res;
}
…
long res=Multiply(2,4,10,5,612);
```

Parameters

- Parameters aanroepen met hun naam.
  - Gebruik <naam>:<value> => bijvoorbeeld x:10
  - Volgorde van de parameters is in dit geval niet belangrijk

```
public int Sum(int a, int b) { return a + b; }
int s = Sum(b:5,a:2);
```

  - Dit is vooral handig bij optionele parameters!

Null Operators

- Null coalescing operator: ??

```
string empty = null;
string txt = empty ?? "leeg";    => txt = “leeg”
```

ⓘ Wanneer we enkel iets willen toekennen indien het type null is kunnen we ??= gebruiken

```
string empty = null;
empty??="leeg";          => empty = “leeg”
```

- Null conditional operator (of ‘Elvis’ operator): ?
  - Als het object waarvan een functie wordt aangeroepen null is, wordt er null teruggegeven ipv dat er een NullException wordt geworpen.

```
string empty = null;
string upperTxt=empty?.ToLower();
```

  - De ultieme ontvanger moet wel een null kunnen ontvangen:

```
int len = txt?.Length;        => FOUT !
```

➢ (* `int? len = txt?.Length;` is wel toegelaten, we behandelen int? later in de cursus bij boxing/unboxing)

  - Void funties zijn wel toegelaten, die worden in dit geval genegeerd.

```
SomeClass myClass = null;
myClass?.MakeSomething(); => wordt genegeerd door het systeem
```

Durf beslissingen te nemen!

- Met if:
  - De if wordt uitgevoerd als het statement true is:

```
if(2 > 1) Console.WriteLine("true");

if(2 > 1 || 1 < 2) {
    Console.WriteLine("true");
}
```

Met haakjes is het duidelijker!

```
if((2 > 1) || (1 < 2)) {
    Console.WriteLine("true");
}
```

  - En anders … else

```
if(isTrue) {
   Console.WriteLine("true");
} else {
   Console.WriteLine("false");
}
```

  - Een if kan ook genest worden

```
if(isExit) {
   Console.WriteLine("true");
} else {
   if(true) {
      Console.WriteLine("Een beetje true");
   } else {
      Console.WriteLine("false");
   }
}
```

Durf beslissingen te nemen!

- Met een switch:
  - Met een **switch** kan je meerdere if statements combineren in een mooie structuur:

```
switch(cmd?.ToLower()) {
  case "exit":
    isExit=true;
    break;
  case "clear":
    Console.Clear();
    break;
  default:
    Console.WriteLine("Ongekend commando");
    break;
}
```

  - Spelregels:
    - De waarde van elke case moet een **constante** zijn.
    - De switch kan dus gebruikt worden met **string**, **integers**, **char** en **enum.**
    - Op het einde van elke case moet een **jump** statement gebruikt worden om de volgende stap aan te duiden => **break** is het meest logische.
    - Jump statements kunnen gebruikt worden: **break**, **goto case *x*, *goto default*, *return.***

Labo: Condities

- Functionaliteit:
  - In plaats van resultaten krijgen de studenten vanaf nu een letter. Van A – E. Schrijf een functie die een verduidelijking van de letter geeft:
    - A uitstekend
    - B goed
    - C net voldoende
    - D ondermaats
    - E rampzalig
- Test cases:
  - Input:
    1. A
    2. C
    3. E
  - Output:
    1. Uw resultaat is uitstekend
    2. Uw resultaat is net voldoende
    3. Uw resultaat is rampzalig

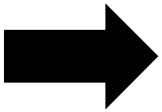
Iteraties

- Het ‘for’ statement:
  - De initialisatie van een for loop bevat 3 belangrijke elementen:
    - Initialisatie
    - Conditie
    - Iteratie

for(int i=0; i < 4 ; **i++**){…}

  - Initialisatie: wordt aangeroepen voor de loop wordt uitgevoerd en initialiseert één of meerdere variabelen.
  - Conditie: bij het begin van elke iteratie wordt dit statement geëvalueerd.
  - Iteratie: wordt uitgevoerd na elke iteratie.

```
for(int i = 2, plus = 1;i < 20;i += plus) {
  plus = i;
  Console.WriteLine("plus = " + i);
}
```



```
plus = 2
plus = 4
plus = 8
plus = 16
```

Iteraties

- while
  - De iteratie wordt uitgevoerd zolang de conditie true is.
  - De expressie wordt getest voor elke aanvang van de loop.

```
while(!isExit) {
    ...
}
```

  - Pas op voor oneindige loops!
- do … while
  - Hetzelfde maar omgekeerd: de expressie wordt op het einde van de loop geëvalueerd.

```
do {
    ...
} while(!isExit);
```

Iteraties

- De foreach loop
  - Iteraties over elk element in een ‘enumerable’ object, zoals array, lijsten, string.
  - De foreach evalueert elk element die dit enumerable object bevat.

```
string content = "hottentottententententoonstelling";
foreach(char ch in content) {
   Console.Write(ch + " ");
}
```

```
h o t t e n t o t t e n t e n t e n t e n t o o n s t e l l i n g
```

Jump statements

- break
  - Beëindigt de uitvoering van een iteratie of een switch statement.

```
while(true){
   if(DateTime.Now.Hour > 10) break;
}
```

- continue
  - Slaat in een iteratie de rest van het statement over en begint onmiddellijk aan de volgende iteratie.
- Return
  - Verlaat de functie en geeft eventueel een waarde terug.
- goto
  - Springt naar een label.

```
begin:
if(DateTime.Now.Minute == 10) goto begin;
```

Array

- Een array bevat een vooraf bepaald aantal elementen.
- De elementen worden in een aaneensluitend deel van het geheugen gepositioneerd. Dit zorgt voor een uitstekende performantie.
- Een array wordt gedeclareerd door het <type> + [] + <naam>
  - Int [] myArray;
- Wanneer een instantie wordt aangemaakt moet de exacte grootte worden meegeven.
  - Int [] myArray=new int[20]; => Een array van 20 integers wordt aangemaakt.
- De grootte van een array kan nadien niet meer worden gewijzigd !
- Bij de creatie van een array wordt elk element geïnitialiseerd met hun standaard waarde (0 voor int, null voor reference types).
- We kunnen een array initialiseren met een expressie:
  - `int[] myArray = { 0,1,2,3 };`

Labo: Array

- Functionaliteit:
  - Schrijf een functie die via de console een serie van getallen aanvaart in tekst vorm en zet die om naar een array van int of double, afhankelijk of deze komma getallen bevatten.
- Test cases:
  - Input:
    1. “10”, ”6”, ”44”, ”1”, ”10200”
    2. “1”, ”2.5”, ”5”, ”11”, ”200”
    3. “9”, ”3”, ”84”, “varken”, ”14”, ”610”
  - Output:
    1. Return = Int [] {10, 6, 44, 1, 10200}
    2. Return = double[] {1.0, 2.5, 5.0, 11.0, 200.0}
    3. Return = null

Multidimensionale arrays

- Rectangular
  - Wanneer we de multidimensionale array bij de initialisatie al onmiddellijk een vaste lengte geven

```
int[,] myMatrix = new int[3,4];
```

  - De **GetLength()** functie van een array stelt ons in staat om de exacte lengte van alle arrays op te vragen.

```
for(int x = 0;x < myMatrix.GetLength(0);x++) {
   for(int y = 0;y < myMatrix.GetLength(1);y++) {
      myMatrix[x,y] = x + myMatrix.GetLength(0) + y;
   }
}
```

  - Ook een multidimensionale array kan met een expressie aangemaakt worden:

```
int[,] myArray = { { 0,1,2,3 },{ 2,1,0,3 } };
```

Multidimensionale arrays

- Jagged arrays
  - Elke dimensie binnen de array kan verschillend van grootte zijn.
  - We creëren voor elke dimensie aparte haakjes.
  - De grootte van de dimensies kunnen bepaald worden bij de initialisatie.

```
int[][] myMatrix = new int[3][];
for(int x = 0;x < myMatrix.GetLength(0);x++) {
   myMatrix[x] = new int[2 + x];
   for(int y = 0;y < myMatrix.GetLength(1);y++) {
     myMatrix[x][y] = x + myMatrix.GetLength(0) + y;
   }
}
```

  - Ook een jagged array kan met een expressie aangemaakt worden, let op de verschillen met een rectangular array.

```
int[][] myArray = { new int[] { 0,1,2,3 },new int[] { 2,1,0,3 },new int[] { 1,2 } };
```

Labo: Multidimensionale array

- Functionaliteit:
  - Maak een functie die een array van strings aanvaard. Die strings kunnen zinnen of uitspraken bevatten. Splits die zinnen in woorden en geef een jagged array terug die per zin de woorden bevat
- Test cases:
  - Input:
    1. String[] { “Op de top kun je geen lange wandelingen maken”, “Het leven is niet te kort maar we beginnen te laat”}
  - Output:
    1. {{“Op”,“de”,“top”,“kun”,“je”,“geen”,“lange”,“wandelingen”,“maken”},{“Het","leven","is","niet","te","kort","maar","we","beginnen","te","laat”}}

Collections

- List<T>
  - T staat voor Type. Hierin kunnen we het type van onze list specifiëren. Dit noemt men een generieke lijst => C# generics
    - `List<string> myList = new List<string>();`
  - Anders als in een array kunnen we elementen toevoegen en verwijderen uit de lijst!
  - We voegen elementen toe met de Add() functie:
    - myList.Add("nieuwe string");
  - Het List object heeft ook een indexer. We kunnen de lijst sequentieel aflopen:

    ```
    string txt = myList[1];
    ```
  - Elementen kunnen verwijderd worden met de functie RemoveAt() => positie

    myList.RemoveAt(1);
  - Het is mogelijk om een array van elementen toe te voegen:

    ```
    string[] strArray = { "element1","element2","element3" };
    myList.AddRange(strArray);
    ```

Collections

- Dictionary<TKey,TValue>
  - Net zoals in de List<T> werkt een Dictionary ook met generics. Toch enkele grote verschillen:
    - Dictionary vraagt 2 types, namelijk een sleutel (Tkey) en een waarde (Tvalue).
    - Men kan een waarde vinden door de gepaste sleutel te zoeken met de indexer [TKey].
    - Een sleutel moet uniek zijn, de waarde niet.

```
static void Main(string[] args) {
  Dictionary<int,string> myDictionary = new Dictionary<int,string>();
  myDictionary.Add(0,"hallo");
  myDictionary.Add(0,"iedereen"); // System.ArgumentException -> Er mogen geen 2
                                                    sleutels voorkomen met dezelfde waarde
  myDictionary.Add(1,"iedereen"); // Toegelaten, waarden mogen hetzelfde zijn
  Console.WriteLine(myDictionary[1]);  // output op scherm : iedereen
}
```

Collections

- Iteratie van een dictionary
  - Omdat een dictionary meerdere elementen bevat, namelijk een sleutel en een waarde, wordt dit paar ook teruggegeven bij een iteratie: KeyValuePair
  - KeyValuePair bevat een property ‘Key’ en een property ‘Value’

```
static void Main(string[] args) {
  Dictionary<int,string> myDictionary = new Dictionary<int,string>();
  myDictionary.Add(0,"hallo ");
  myDictionary.Add(1,"iedereen, ");
  myDictionary.Add(3,"gegroet!");
  foreach(KeyValuePair<int,string> member in myDictionary) {
    Console.WriteLine(member.Value);  // output op scherm : hallo iedereen, gegroet!
  }
}
```

Andere handige datatypes

- DateTime
  - Een ingebouwde class die een punt in de tijd weergeeft.
  - Het is een value type.
  - Cultuur afhankelijk -> current culture.
  - De tijd wordt intern bijgehouden in een long '**Ticks**' die de verstreken nanoseconden (per 100 nanoseconden) telt sinds het jaar 1.
  - Men kan de huidige datum/tijd opvragen door de property '**Now**' aan te roepen of de UTC tijd door 'UtcNow'

```
static void Main(string[] args) {
   DateTime now = DateTime.Now;
   DateTime utc = DateTime.UtcNow;
}
```

  - **DateTimeOffset** gebruikt men als men relatief met de UTC werkt.

Andere handige datatypes

- TimeSpan
  - Men gebruikt TimeSpan om het verschil tussen 2 punten in tijd aan te duiden.

```
static void Main(string[] args) {
  DateTime now = DateTime.Now;
  DateTime utc = DateTime.UtcNow;
  TimeSpan dif = now - utc;
  Console.WriteLine(Math.Round(dif.TotalMinutes));
}
```

Creëer class om een drankautomaat te beheren

- Oefening :
  - Functionaliteit:
    - De automaat heeft een aanbod aan dranken die elk een andere prijs (kunnen) hebben.
    - We hebben een automaat die enkel Euro munten kan teruggeven maar die wel biljetten aanvaard.
    - De gebruiker kiest een drank uit een genummerde lijst en betaalt
    - De machine registreert hoeveel van elke drank nog beschikbaar is.
  - Maak een console app om de werking te simuleren
  - Test cases:
    1. Keuze is Cola : Kostprijs 2.20€. Beschikbaar: 4. Betaald: 5€
       - Wisselgeld: 1 x 2€ + 1 x 50 c + 1 x 20c + 1x 10c + Cola. Nog beschikbaar: 3
    2. Keuze is water: Kostprijs 1.25€. Beschikbaar: 1. Betaald: 10€.
       - Wisselgeld: 4 x 2€ + 1 x 50 c + 1 x 20c + 1x 5c Beschikbaar: €Niet meer beschikbaar
    3. Keuze is fruitsap: Kostprijs: 2.10. Beschikbaar: 0. Betaald: 5€
       - Wisselgeld: 2x2€ + 1x1€. Boodschap: Uw keuze is niet meer beschikbaar!

Constanten

- Is een ‘static field’ dat nooit veranderd kan worden.
- Een constante wordt door de compiler rechtstreeks als waarde in de IL geplaatst.
- Een constante moet één van volgende ingebouwde types zijn: integers, reals, bool, char of string.
- De creatie van een constant veld gebeurt met het keyword ‘**const**’

```
public const long DefaultLength = 100;
```

- We kunnen ev. ook een constante declareren in een functie.

Enum: enumeratie of opsomming

- Een enum is een value type dat toelaat om een groep constanten te creëren.

```
public enum Colors { red , green, blue };
```

- Elke enum heeft onderliggend een int type.
- Het is ook mogelijk om elk lid een aangepaste waarde te geven tijdens de declaratie.

```
public enum Colors { red = 1, green = 2, blue = 3 };
```

- Het is ook mogelijk om het datatype van een enum te wijzigen in een alternatief integer type.

```
public enum Colors :byte { red , green = 2, blue  };
```

Enum: enumeratie of opsomming

- Conversie
  - Men kan een enum instantie **expliciet** casten **van** en **naar** het **onderliggende type**:

```
static void Main(string[] args) {
  Colors myColor = Colors.red;
  byte colorVal = (byte) myColor;
  Console.WriteLine(colorVal);  // 1
  Console.WriteLine(myColor);   // red
  colorVal += 1;
  myColor = (Colors)colorVal;
  Console.WriteLine(myColor);   // green
}
```

  - Indien men een verkeerde cast doet zal de CLR **geen** Exception genereren maar aan het onderliggende type gewoon de waarde toekennen.

```
  colorVal = 255;
  myColor = (Colors)colorVal;
  Console.WriteLine(myColor);   // 255 !!!
```

Enum: enumeratie of opsomming

- Flags
  - Indien men waarden van een enum wil combineren kan men **flags** gebruiken.
  - Men declareert de waarden in **machten** van **2** om **binair** te **vergelijken.**

```
[Flags]
public enum GridLines { Left = 1, Right = 2, Top = 4, Bottom = 8 };
…
static void Main(string[] args) {
  GridLines gridConfig = GridLines.Left | GridLines.Right;
  if((gridConfig & GridLines.Left) == GridLines.Left) { Console.WriteLine("Grid left"); }
  if((gridConfig & GridLines.Top) == GridLines.Top) { Console.WriteLine("Grid top"); }
  // OUTPUT : Grid left
  Console.WriteLine(gridConfig); // Left, Right
}
```

  - Het is ook mogelijk om standaard enkele waarden te **combineren** in de declaratie:

```
[Flags]
public enum GridLines { Left = 1, Right = 2, Top = 4, Bottom = 8, LeftAndRight = Left | Right };
```

Labo: werknemersbestand

- Ontwerp de nodige klassen om een werknemersbestand aan te maken.
- Voor elke werknemer hebben we volgende data nodig:
  - De voor- en achternaam van de werknemer
  - Datum in-dienst
  - Adresgegevens, telefoon, email,…
- Werknemers kunnen in-dienst genomen worden of uit-dienst gaan.
- Let op, sommige werknemers kunnen meerdere email adressen hebben of telefoonnummers (mobiel, thuis, werk,…)
- Maak een CLI aan waar we de lijst van werknemers kunnen opvragen, een werknemer kunnen zoeken op naam, een werknemer kunnen verwijderen uit de lijst en een nieuwe naam toevoegen aan de lijst.

Exceptions

- Exceptions worden geworpen bij runtime fouten.
- We kunnen deze opvangen in een try – catch blok.
- We kunnen een ‘overload’ van het catch blok maken om de juiste exception op te vangen.

```
try {
  …
} catch(NullReferenceException nullRefEx) {
  …
} catch(NotSupportedException  notSupportedEx) {
  …
} catch(Exception generalEx) {
  …
}
```

Het gebruik van exceptions

- Enkel in onvoorziene omstandigheden! Een exception is zeer kostelijk voor de CLR.
- Wanneer exceptions opvangen ?
  - Om het programma een mogelijkheid te bieden om een onvoorziene fout te herstellen.
  - Als we de fout willen loggen of debuggen om deze daarna de exception terug te werpen (**throw**).

```
throw generalEx; of throw new Exception();
```

  - Als laatste redmiddel om het programma niet te laten vastlopen.
- We kunnen alle exceptions opvangen door de basis klasse te gebruiken (Exception).
- Vanaf C#6 kunnen we filters specifiëren om een specifieke fout op te vangen met het keyword ‘when’

```
} catch(OverflowException  overflowEx) when (overflowEx.GetBaseException()!=null) {
```

finally

- We kunnen het try{…}catch(…){…} blok uitbreiden met een **finally** statement.
- Het **finally** statement wordt steeds uitgevoerd **bij het verlaten** van het try blok, ook als er zich geen fout voordoet.
- We gebruiken het finally block vooral om resources vrij te geven, zoals een lock of stream.

```
FileStream stream =null;
try {
  stream = File.Open("myfile",FileMode.Open);
  …
} catch(Exception) {.
  …
} finally {
  stream.Flush();
  stream.Close();
}
```

Statische functies

- Standaard functies werken op de instantie van een class. Deze worden **‘instance members’** genoemd en kunnen pas aangesproken worden als er een instance van de class wordt aangemaakt.
- **Static members** werken rechtstreeks op het type zelf en hebben geen instantie van de class nodig om gebruikt te worden. Dit heeft als gevolg dat ze niet rechtstreeks ‘instance’ leden kunnen aanspreken.

```
public class StaticDemo {
   int myVar = 0;
   public void SetVar(int var) {
     myVar = var;
   }
   public static void SetVarStatic(int var) {
     myVar = var;                    // => Error !!!!
   }
   public static void SetVarStatic(StaticDemo parent,int var) {
     parent.myVar = var;                    // => OK !!!!
   }
 }
```

Statische functies

- Een statische functie kan rechtstreeks worden aangeroepen op het type:

```
static void Main(string[] args) {
   StaticDemo demo = new StaticDemo();
   StaticDemo.SetVarStatic(demo,5);
 }
```

- We kunnen ook statische fields declareren. In dit geval is de waarde gebonden aan het type en niet aan een instantie van het object!

```
public class StaticDemo {
   static int myVar = 0;
   public static int MyVar { get { return myVar; } set { myVar = value; } }
}
…
static void Main(string[] args) {
     StaticDemo.MyVar = 4;
}
```

Delegates

- C# (en .Net) laat toe dat we niet alleen referenties naar types maar ook referenties naar functies in types kunnen specifiëren.
- Dit laat toe dat een type andere objecten kan contacteren zonder dat deze objecten moeten gekend zijn.
- Eerst moeten we de functie declareren zodanig dat de betrokken partijen weten welke vorm dat die functie heeft. Zo een afspraak wordt voorafgegaan door het keyword '**delegate**'.

```
public delegate string CallbackFunction(int num,string txt);
```

- De class kan daarna de delegate functie declareren als een field of een property. Omdat het over een reference gaat kan de waarde ook null zijn.

```
public class CallbackDemo {
  public CallbackFunction myCallback = null;
  public CallbackDemo(CallbackFunction cb) { myCallback = cb; }
  public void DoSomeWork() { if(myCallback != null) {
    Console.WriteLine(myCallback(20,"Charel"));
  }
}
```

Delegates

- Daarna kan een ander type een functie declareren die dezelfde lay-out heeft als de declaratie.

```
public class PrintClass {
  public static string Print(int score,string name) {
    return $"{name} heeft {score} punten!";
  }
}
```

- Bij het creëren van de instantie kan de ‘delegate’ meegegeven worden en daarna door de ontvangende class worden aangeroepen.

```
static void Main(string[] args) {
      CallbackDemo cbDemo = new CallbackDemo(PrintClass.Print);
      cbDemo.DoSomeWork(); // Console output ->  Charel heeft 20 punten!
}
```

- Een ‘delegate’ is zeer geschikt om ‘plug-in’ functies te maken!

Delegates

- Multicast
  - Het is mogelijk om meer dan 1 delegate toe te voegen aan de delegate reference.
  - Met += en -= kan men meerdere delegates toevoegen of verwijderen.
  - Bij de aanroep van de delegate functie worden alle ingevoegde delegates aangeroepen.

```
CallbackDemo cbDemo = new CallbackDemo(PrintClass.Print);
cbDemo.myCallback += SuperPrintClass.Print;
cbDemo.DoSomeWork(); // Beide callback functies worden aangeroepen
```

  - OPGEPAST! Indien de delegate een waarde teruggeeft zal enkel de waarde van de laatst aangeroepen delegate worden gebruikt. De rest van de delegate functies worden nog wel aangeroepen.

Labo

- Pas je CLI basis klasse aan zodat een delegate wordt aangeroepen indien het standaard commando niet gekend is.
  - Zorg dat een externe klasse een delegate kan toekennen waarin de nodige informatie wordt uitgewisseld.
- Maak in je werknemersbestand een delegate functie aan waarin je commando's kan integreren in de CLI basis.

Labo

- Maak een klasse ‘recept’ aan.
  - Het recept moet het volgende bevatten:
    - Naam
    - Type : Voorgerecht, soep, hoofdgerecht of dessert. Gebruik hiervoor een enum
    - Ingrediënten
    - Beschrijving van de werkwijze
  - De klasse moet een functie hebben die een recept op het scherm afdrukt
  - De klasse moet de mogelijkheid bieden om delegate functies toe te voegen die worden aangeroepen nadat het recept op het scherm verschijnt
- Maak enkele klassen aan met persoonsnamen.
  - De personen moeten een delegate callback functie declareren waarin ze hun eigen aanpassingen aan het algemeen recept kunnen toevoegen die onderaan het scherm wordt afgedrukt.
    - De naam van de persoon moet ook worden afgedrukt samen met de aanpassingen.
- Maak minstens 2 personen of meer aan.