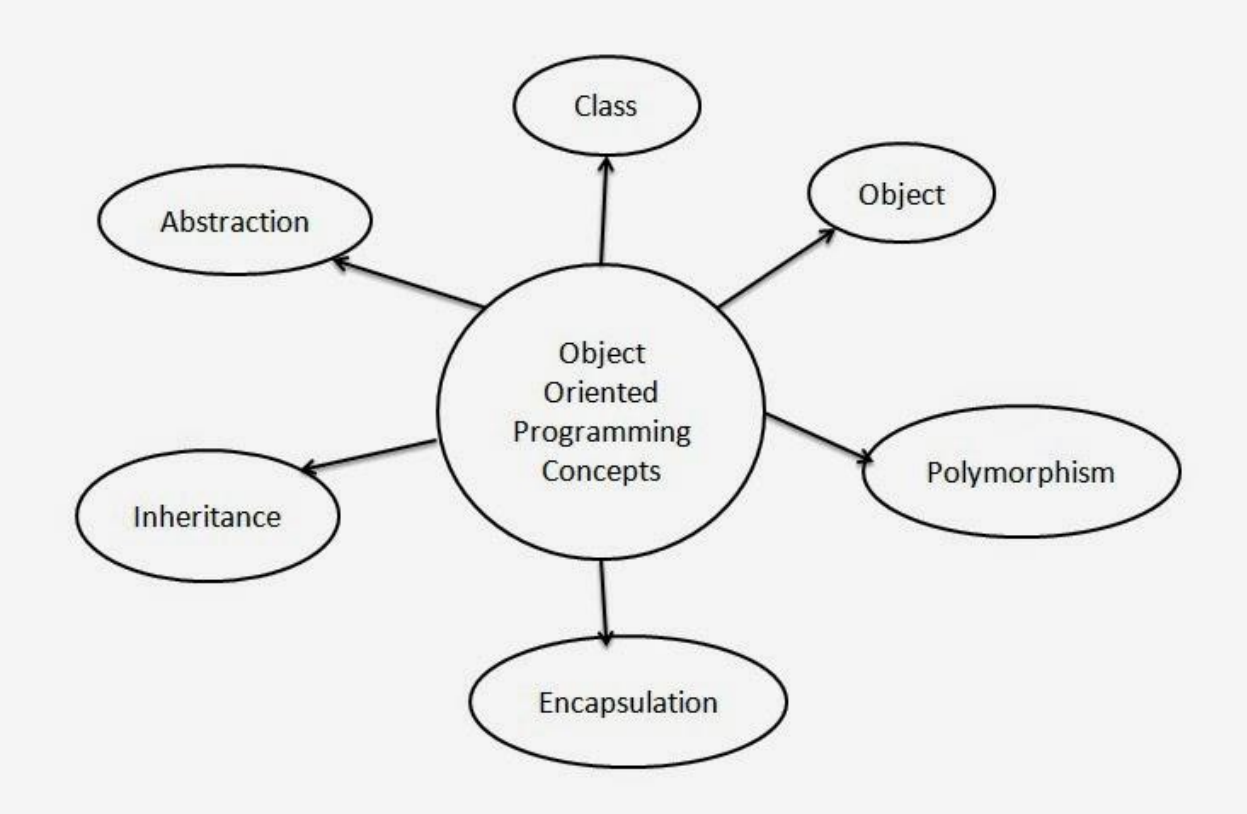
# C# Classes en Objects

In de vorige lessen zijn de termen class en object al enkele keren genoemd, maar er is nog niet concreet over deze termen gesproken. Dat gaat in deze les gebeuren.




Bij een **Object-ge-Oriënteerde Programmeertaal (OOP)** gaat het om de objecten en diens eigenschappen. Objecten zijn concrete dingen in een wereld. Bijvoorbeeld de zwarte ronde tafel is een ander object dan de rechthoekige metalen tafel.

Een huis in de straat is een concreet object, maar de plattegrond (blauwdruk tekening) van dit huis dat wordt een klasse ofwel class genoemd. Heb je eenmaal de blauwdruk dan zijn meerdere van het zelfde object te maken. Denk maar aan een reeks identieke appartementen.

Doormiddel van OOP is het mogelijk om ingewikkelde systemen te implementeren door deze op te splitsen in kleinere en eenvoudigere objecten die met elkaar samenwerken.

Neem bijvoorbeeld het object auto. Dit object bestaat uit een groot aantal objecten. De chassis, de motor, de wielen, etc. Je kunt zelfs nog verder detailleren, bijvoorbeeld een wiel bestaat uit een band, velg, moeren, etc.

# C# Classes en Objects


Een ander belangrijk voordeel van het werken met OOP is **inheritance** (*overerving*). Overerving zorgt ervoor dat eigenschappen kunnen worden overgedragen aan andere objecten van hetzelfde type. Denk aan een zoon die bepaalde eigenschappen van zijn vader en moeder heeft overgenomen. Ook kun je het zien als een vorm van classificatie. Neem bijvoorbeeld een kat en een hond. Twee verschillende dieren, maar die toch ook een aantal overeenkomsten hebben. Beiden zijn viervoeters, beiden hebben een vacht. Beiden horen tot de klasse dieren. Je kunt zeggen kat en hond zijn subklassen van de klasse dieren.

Een andere belangrijke eigenschap van OOP is de mogelijkheid van objecten om hun inwendige details te verschuilen. Dit soort objecten worden vaak "**Black boxes**" genoemd.

# C# Classes en Objects


## Objecten definiëren

Wanneer we een object definiëren zijn er over het algemeen 3 zaken waar je aan moet denken:

| | |
|---|---|
| **Data** | Welke variabelen of andere objecten heeft het nodig om zijn functie uit te kunnen voeren. |
| **Methoden** | Welke methoden (functies) worden gedefinieerd voor publiek gebruik? Welke methoden voor intern gebruik? |
| **Relaties** | Wat is de interactie met andere objecten? In hoeverre heeft het object overeenkomsten met andere dat het als subklasse kan worden gezien of kan erven van andere. |

## Data

De meeste objecten hebben een of andere vorm van data. De data kan bestaan uit **eigenschappen (properties)** of andere objecten.

In een muziekprogramma zou je bijvoorbeeld het object lied kunnen hebben dat de verschillende liederen die gespeeld kunnen worden representeert.

Het object lied zou de volgende eigenschappen kunnen:

- Titel
- Artiest
- Groep
- Jaar
- Album naam
- Lengte
- Genre

Lied zou ook ander objecten kunnen bevatten zoals:

- Noten – Een lijst met de muzieknoten van het lied.
- Liedtekst (Lyrics) – De tekst waar het lied uit bestaat.

Het is belangrijk om goed te controleren welke rechten een extern programma heeft over de data van een object. Elke eigenschap kan worden gedefinieerd als ***public*** of ***private***.

Externe componenten hebben vrije toegang tot public data. Private data is alleen toegankelijk voor methoden van het object zelf.

# C# Classes en Objects

## Methods/Methoden

Een object heeft methoden nodig om acties te kunnen uitvoeren. Net als met data kunnen methoden public, private of protected gedefinieerd worden.

*Public* methoden zijn functies die toegankelijk zijn voor iedereen. Deze methoden kunnen data in een object wijzigen.

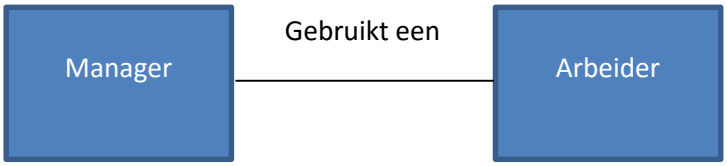
*Private* methoden zijn alleen toegankelijk voor het object zelf. Private methoden kunnen worden aangeroepen door de publieke en privé methoden van het object.

Protected methoden zijn toegankelijk voor alles en iedereen binnen de klasse en afgeleiden ervan, maar alle anderen worden geweigerd.

## Relationships/Relaties

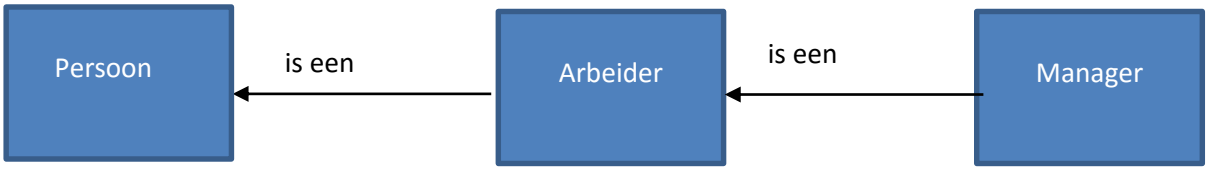
Objecten hebben interacties met andere objecten in het programma. Er zijn drie hoofd relaties die twee objecten met elkaar kunnen hebben.

| | |
|---|---|
| **Uses-a/gebruikt een** | **Object A "gebruikt" Object B door publieke methoden van Object B aan te roepen.** |
| **Has-a/heeft een** | **Object A "heeft een" instantie van Object B als interne data.** |
| **Is-a/is een** | **Object A "is een" instantie van Object B, dit betekent dat Object A erft van, of is een subklasse van Object B. Bijvoorbeeld, een personenauto object "is een" vervoersmiddel object.** |

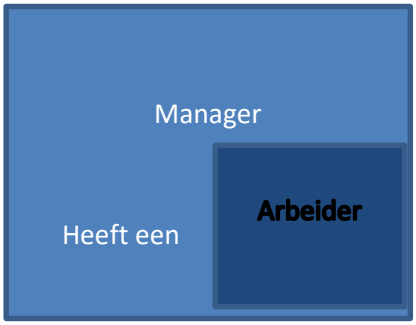


# C# Classes en Objects

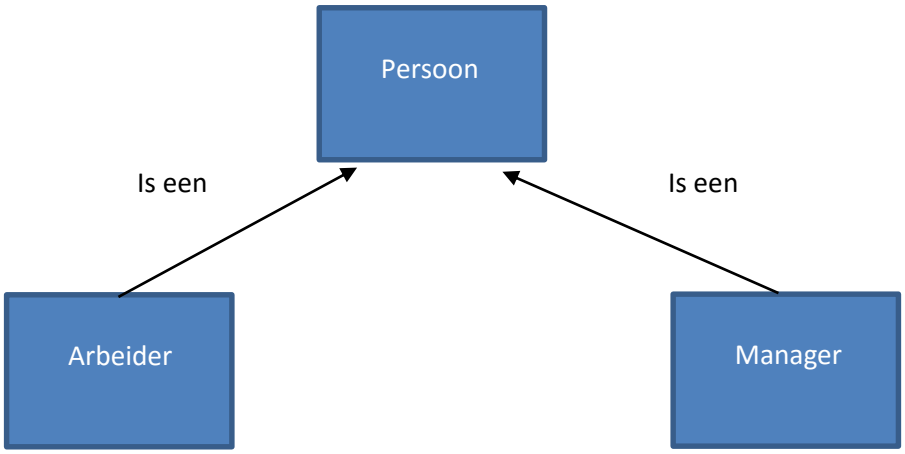
Het Manager object geeft opdrachten aan het Arbeider object




Het Person object bevat algemene eigenschappen zoals naam en leeftijd. Het Arbeider object erft deze van het persoon object en voegt bijvoorbeeld toe werk. Het Manager object erft van arbeider object en voegt weer andere specifieke eigenschappen toe.




Het Manager object bezit het Arbeider object, daar de arbeiders aan de manager moeten rapporteren.




Het kan zijn dat Manager en Arbeider niet veel gemeen hebben, je kunt dan beide van Person object laten erven.

# C# Classes en Objects

**Concept van een Klasse (Class)**

Een klasse is een simpel abstract model dat gebruikt wordt om nieuwe data types te definiëren. Een klasse kan een combinatie van ingekapselde data (velden of variabelen) bevatten.

Een klasse in C# wordt als volgt gedeclareerd:

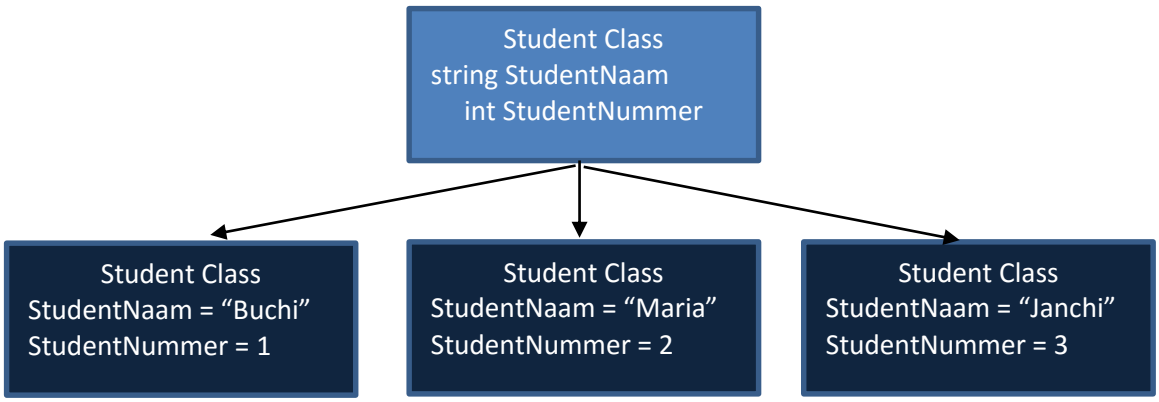
```
class MijnKlasse
{
    // velden, operaties en eigenschappen komen hier
}
```

MijnKlasse is de naam van de klasse of nieuwe data type dat we definiëren.

Objecten

Een klasse is een abstract model. Een object is een concrete realisatie ofwel een instantie gemaakt van specificaties van de klasse. Een object wordt in het geheugen gemaakt met het sleutel woord "new". Dit gebeurt als volgt:

```
MijnKlassecte mijnObjectReferentie = new MijnKlasse();
```




**Fields/Velden**

Velden bevatten de data van de klasse. Velden kunnen impliciete data bevatten, objecten van een klasse, **enumerations**, **structs** of **delegates**. Je kunt velden initialiseren met een initiële waarde. Als dat niet gedaan wordt dan worden ze geïnitialiseerd met hun standaard waarden.

# C# Classes en Objects

In het voorbeeld hieronder is er een klasse Student gedefinieerd:

```
class Student
{
    // velden in de klasse Student
    string      naam;
    int         leeftijd;
    int         cijferInformatica;
    int         cijferWiskunde;
    int         cijferEconomie;
    int         totaalPunten = 300;     //initioalisatie
    int         behaaldePunten;
    double      percentage;
}
```

Standaard waarden voor verschillende data typen staan in de onderstaande tabel:

| Data type | Standaard waarde |
|---|---|
| int | 0 |
| long | 0 |
| float | 0.0 |
| double | 0.0 |
| bool | False |
| char | '\0' (null character) |
| string | "" (empty string) |
| Objects | null |

**Methods**

Methoden (methods) zijn operaties die uitgevoerd worden op data. Een methoden kan waarden mee krijgen als parameters en kan een waarde van een bepaald data type terug sturen. De structuur van een Method is als volgt:

```
<return types> <name of method>(<data type> <identifier>, …)
{
    // body of method
}
```

# C# Classes en Objects

**Instantie van een klasse (class instance)**

In C# moet men een instantie maken van een klasse. Hiermee wordt bedoeld dat er een concreet object gemaakt wordt van de klasse. Dit gebeurt met het sleutelwoord ***new***.

```
Student deStudent =  new Student();
```

Je kunt ook de referentie declareren en in een andere stap een object toekennen.

```
Student deStudent;
deStudent = new Student();
```

**Toegang tot leden (members) van een klasse**

Je krijgt toegang tot de verschillende members van een klasse met de '.' operator.

```
Student deStudent = new Student();
deStudent.cijferInformastica = 95;
deStudent.BerekenTotaal();
Console.WriteLine(deStudent.vindtPunten);
```

Laten we kijken naar een uitgewerkt voorbeeld waar de Student klasse wordt gemaakt met enkele velden en methoden en deze dan instantiëren in the Main() methode.

```
using System;
namespace StudentPunten
{
    // Definiëer een klasse om Studenten gegevens te verwerken
   Class Student
   {
        // Velden
        string        naam;
        int           leeftijd;
        int           cijferInformatica;
        int           cijferWiskunde;
        int           cijferEconomie;
        int           totaalPunten = 300;      //initioalisatie
        int           behaaldePunten;
```

```
        double       percentage;

        // methoden
        void BerekenTotaalPunten()
        {
             vindtPunten = cijferInformatica + cijferWiskunde +
cijferEconomie;
        }

        void BerekenPercentage()
        {
             percentage = (double) vindtPunten / totaalPunten * 100;
        }

        double VindtPercentage()
        {
              return percentage;
        }

        // Main method
        static void Main()
        {
            // Maak een nieuwe instantie van Student
            Student st1 = new Student();

            // geef de velden een waarde
            st1.naam = "Juancho";
            st1.leeftijd = 18;
            st1.cijferInformatica = 75;
            st1.cijferWiskunde = 95;
            st1.cijferEconomie = 80;

            // methodes aanroepen
            st1.BerekenTotaalPunten();
            st1.BerekenPercentage();
            double st1Percentage = st1.VindtPercentage();

            Console.WriteLine("{0} van de {1} jaren hebben {2}%
punten", st1.naam, st1.leeftijd, st1.percentage);

        }
    }
}
```

# C# Classes en Objects


**Toegangs aanpassers (Access Modifiers)**

In de Student klasse heeft iedereen toegang tot the velden. Dus iedereen zou de waarden van de velden kunnen wijzigen. Met access modifiers is het mogelijk in C# om de toegang te bepreken.

| Access Modifier | Omschrijving |
|---|---|
| private | private members zijn alleen toegankelijk binnen de class waarin ze zich bevinden. |
| protected internal | Deze zijn toegankelijk binnen het huidige project of door types die erven van de type waartoe ze behoren. |
| internal | Deze zijn alleen toegankelijk binnen het huidige project. |
| protected | Zijn toegankelijk binnen de bevattende klasse en types die erven van de bevattende klasse. |
| public | Iedereen die deze ziet heeft toegang tot ze. |

In OOP wordt altijd geadviseerd om de velden als private aan te geven en alleen bepaalde methods aan de gebruiker vrij te geven door deze public te maken.

**Properties**

Je zult je misschien afvragen als we alle velden als private declareren, hoe krijg je dan toegang tot de velden. In C# gebeurt dit doormiddel van **properties**.
Om toegang te krijgen tot deze private velden worden er public methods gemaakt die "**getters**" (om een waarde op te halen) heten en "**setters**" (om een waarde toe te kennen) aan zo'n private veld.

Als we een private veld hebben:

```
private string naam;
```

Dan zijn de getters en setters als volgt:

```
public string GetNaam()
{
    return naam;
}
public void SetNaam(string deNaam)
{
    naam = deNaam
}
```

Door deze te gebruiken kunnen we de toegang tot de velden beperken. Men zou bijvoorbeeld alleen een getter kunnen maken. Dit heeft tot gevolg dat de gebruiker van de class alleen de waarden kan zien maar niet veranderen.

**Properties syntax**

```
<access modifier> <data type> <name of property>
{
    get
    {
        // some optional statements
        return <some private field>;
    }
    set
    {
        // some optional statements
        <some private field> = value;
    }
}
```

Dus als we een private veld naam hebben

```
private string naam;
```

Dan zien de properties er zo uit

```
public string Naam
{
    get
    {
        return naam;
    }
    set
    {
         naam = value;
    }
}
```

**value** is een sleutelwoord en het bevat de waarde die wordt doorgestuurd wanneer de property wordt aangeroepen.

# C# Classes en Objects

```
Student deStudent = new Student();
theStudent.Naam = "Rachel";
string mijnNaam = theString.Naam;
theStudent.naam = "Iemand die niet Rachel is"; // error
```

Properties zijn context sensitief.

```
deStuden.Naam = "Rachel";
```

De compiler ziet dat de property Naam, aan de linker zijde van de assignement operator is, dus zal de compiler de set {} blok van de properties aanroepen, die "Rachel" als waarde *(value, welke een sleutelwoord is)* doorsturen.

In de volgende regel:

```
string mijnNaam =  deString.Naam;
```

Hier ziet de compiler de property Naam aan de rechter kant van de toekenningsoperator. Nu wordt het get {} blok van de property Naam aangeroepen welke de waarde ("Rachel") van het prive veld naam zal bewaren in het lokale veld naam.

De laatste regel:

```
theStudent.naam = "Iemand die niet Rachel is"; // error
```

Zal een compiler fout generen omdat het veldnaam privé is gedeclareerd in de class declaratie.

**Opmerkingen:**

- Properties hebben geen lijst van parameters (argumenten)
- set, get en value zijn sleutel woorden in C#
- Het datatype van value is het zelfde als het van de property dat gedeclareerd is wanneer de property wordt gedefinieerd.
- Gebruik **ALTIJD** accolades {} en correcte vorm van inspringen wanneer je properties gebruikt.
- Probeer **NOOIT** om een set {}  of een get {} blok in een enkele regel te schrijven.
- TENZIJ de property alleen toekent en opvraagt van waarden van prive velden zoals:

```
get { return naam;}
set { naam = value; }
```

# C# Classes en Objects


**Static members van een class**

Static members behoren tot de gehele klasse in plaats van tot een individueel object.

Als er bijvoorbeeld een static veld telefoonNummer in de klasse Student is, zal er één enkele instantie van dat veld zijn. Alle objecten van de klasse delen dit enkele veld. Aanpasingen die door één object gemaakt worden aan telefoonNummer zullen worden erkent door de andere objecten in de klasse.

```
class Test
{
    Public static void Main()
    {
        Student st1 = new Student();
        Student st2 = new Student();
        st1.rolNummer = 3;
        st2.rolNummer = 5;
        Student.telefoonNummer = 1234567;
    }
}
```

Hier zie je dat het veld telefoonNummer wordt aangeroepen zonder te refereren naar een object, maar met de naam van de klasse.

**Opmerkingen:**

- Gebruik niet te veel static methods in een klasse, daar dit tegen het principe van object georiënteerd ontwerpen ingaat.
- Je verliest een aantal OO voordelen wanneer je static methods gebruikt.

**Constructors**

Constructors zijn speciale methods. Een Constructor heeft de volgende eigenschappen:

- Het heeft dezelfde naam als de klasse waar het toebehoort.
- Het heeft geen return type.

# C# Classes en Objects

- Het wordt automatisch aangeroepen wanneer een nieuwe instantie of een object van een klasse wordt aangemaakt. Daarom ook de naam Constructor.
- De Constructor bevat initialisatie code voor elk object, zoals het toekennen van standaard waarden aan velden.

Bijvoorbeeld:

```
using System;

Class Persoon
{
    // veld
    private string naam;

    // constructor
    public Persoon()
    {
          naam = "onbekend";
          Console.WriteLine("Constructer aangeroepen …");
    }

    // property
    public string Naam
    {
        get { return naam; }
        set { naam = value; }
    }
}
```

In de Persoon klasse hierboven, hebben een we een privé veld naam, een publieke constructor welke het naam veld initialiseert met de tekst "onbekend" en op het console venster afdrukt dat het is aangeroepen.

# C# Classes en Objects

We maken een andere klasse Test, welke de Main() method bevat en gebruik maakt van de klasse Persoon.

```
class Test
{
    public static void Main()
    {
        Persoon dePersoon = new Persoon();
        Console.WriteLine("De naam van de persoon in object dePersoon is
" + dePersoon.Naam);
        dePersoon.Naam = "Maria";
        Console.WriteLine("De naam van de persoon in object dePersoon is
" + dePersoon.Naam);
    }
}
```

In de Test klasse, is een object van de Persoon klasse gemaakt en da naam van de persoon afgedrukt. Daarna is de waarde van Name veranderd en opnieuw afgedrukt. Het resultaat van het programma is:

```
Constructor called …
De naam van de persoon in object dePersoon is onbekend
De naam van de persoon in object dePersoon is Maria
```

Als de klasse Persoon private was gemaakt in plaats van public dan zou er een fout zijn opgetreden.

De Contructors die tot nu toe getoond zijn gebruikten geen parameters. Er kunnen echter ook Constructors gemaakt worden die wel parameters accepteren.

```
Class Persoon
{
    private string naam;

    public Persoon(string deNaam)
    {
         naam = deNaam;
         Console.WriteLine("Constructer aangeroepen …");
    }
}
```

Nu kan het object van de klasse Persoon alleen gemaakt worden door een tekst mee te geven in de Constructor.

```
Persoon dePersoon = new Persoon("Maria");
```

Als je geen Cnstructor definieert voor je klasse, zal de compiler er een parameter-loze lege Constructor genereren.

**Mehod en Constructor Overloading**

Het is mogelijk om meer dan één Methode te hebben met exact dezelfde naam en teruggeef waarde, maar met een verschillend aantal argumenten (parameters). Dit wordt **method overloading** genoemd. Het is bijvoorbeeld helemaal correct om het volgende te schrijven:

```
class Checker
{
    // 1ste overloaded form
    public bool isStandaardWaarde(bool val)
    {
        if(val == false)
            return true;
        else
            return false;
    }
    // 2de overloaded form
    public bool isStandaardWaarde(int val)
    {
         if(val == 0)
             return true;
         else
             return false;
    } bool
    // 3de overloaded form
    public bool isStandaardWaarde(int intVal, bool booleanVal)
    {
        if (intVal == 0 && booleanVal == false)
            return true;
        else
            return false;
    }
}
```

In de Checker klasse hierboven zijn drie methods gedefinieerd met de naam isStandaardWaarde(). De teruggeef waarde (return type) van allemaal is bool, maar elk heeft een andere parameterlijst. De eerste twee verschillen in datatype van de parameters terwijl de derde verschilt in het aantal parameters. De compiler beslist op basis van het type en aantal parameters, welke van de drie aangeroepen wordt.

```
Checker check = new Checker();
Console.WriteLine(check.isStandaardWaarde(5));        // eerste aanroep
Console.WriteLine(check.isStandaardWaarde(false));    // tweede aanroep
Console.WriteLine(check.isStandaardWaarde(0, true)); // derde aanroep
```

Daar Constructors speciale form van Methods zijn, kunnen we Constructors op dezelfde wijze overloaden.

```
class Persoon
{
    private string naam;

    public Persoon()
    {
        naam = "onbekend";
    }

    public Persoon(string deNaam)
    {
        naam = deNaam;
    }
}
```

**Struct**

De C# struct opdracht is een lichtgewicht alternatief voor een klasse. Het kan bijna hetzelfde als een klasse, maar het is minder "duur" in gebruik dan een klasse. De reden hiervoor is dat nieuwe exemplaren van een klasse op de **heap** worden geplaatst, waar de nieuw geïnstantieerd structs op de **stack** worden geplaatst. Bovendien hoeven er geen verwijzingen naar de structuren gemaakt te worden, zoals met bij de klassen, maar in plaats daarvan wordt er rechtstreeks gewerkt met de instantie van de structuur. Dit betekent ook dat wanneer je een structuur doorgeeft aan een functie, dit gebeurt door waarde door te geven, in plaats van als een referentie.

In het voorbeeld hieronder zie je hoe een struct gebruikt wordt:

```
using System;

class Program
{
   struct Simple
   {
         public int Position;
         public bool Exists;
         public double LastValue;
   };

   static void Main()
   {
         // ... Create struct on stack.
         Simple s;
         s.Position = 1;
         s.Exists = false;
         s.LastValue = 5.5;

         // ... Write struct field.
         Console.WriteLine(s.Position);
   }
}
```

**Recursie**

Je hebt gezien hoe een functie aangeroepen kan worden en hoe een functie een andere functie aanroept.

```
public void functieA()
{
    // wat programma code, onder andere de aanroep naar functieB
   functieB();
}

public void functieB()
{
   // wat programma code
}
```

Recursie *(recursion)* is een manier van programmeren waarbij er een "soort" lus gemaakt wordt door een functie zichzelf te laten aanroepen.

```
public void functieC()
{
    // wat programma code, onder andere de aanroep naar functieC
   functieC();
}
```

De functieC zoals hierboven wordt weergegeven is slecht nieuws want er is geen enkele manier voor functieC om te stoppen om zichzelf aan te roepen. De computer gaat “hangen” ofwel loopt vast tot dat er geen geheugen ruimte meer is en dan volgt er een foutmelding. Hoe lossen we dit op? Bekijk het volgende voorbeeld:

```
public string insertStars(string input)
{
   if (input.Length ==0)
      return “”              // beindig recusie wanneer input leeg is

   // splits input string in 2 delen, de eerste karakter en de rest
   char first = input [0];
   string remainder = input.Substring(1, input.Length – 1);

   //stuur een string terug welke een ster bevat, de eerste letter van de
   //invoerstring en het resultaat van de recursieve aanroep over de rest
   //van de invoer string.
  Return “ * “ + first + insertStars(remainder);
}
```

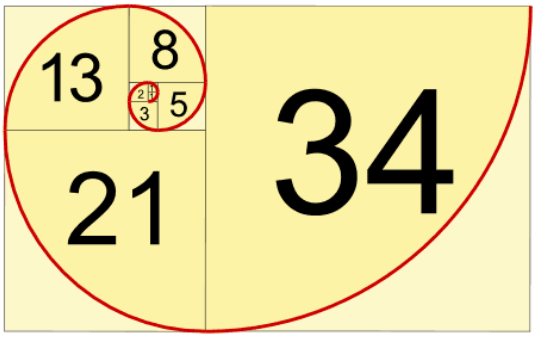
Als we als aanroep gebruiken:

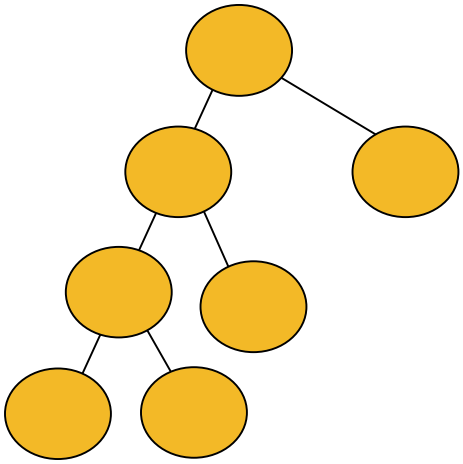
```
String result = insertStars(“Hello”);
```

Dan zal de uitvoer zijn:“* H * e * l * l * o”

| Stap | Functie aanroep | Stap | Terug waarde |
|---|---|---|---|
| 1 | insertStars(“Hello”) | 12 | “* H * e * l * l * o” |
| 2 | insertStars(“ello”) | 11 | “* e * l * l * o” |
| 3 | insertStars(“llo”) | 10 | “* l * l * o” |
| 4 | insertStars(“lo”) | 9 | “* l * o” |
| 5 | insertStars(“o”) | 8 | “* o” |
| 6 | insertStars(“”) | 7 | “” |

Recursie wordt bijvoorbeeld gebruikt voor het sorteren, in sommige wiskundige functies (fibonacchi reeks) en bij het bepalen van een pad in een doolhof of het doorlopen van een binaire boom.



Vaak is recursie een natuurlijke en elegante manier om functies of procedures te definiëren. In een daadwerkelijke implementatie moet men er echter voorzichtig mee zijn. Hoewel recursie soms snel en efficiënt werkt (zoals bij het sorteeralgoritme *Quicksort*), is het ook vaak veel trager dan niet-recursieve implementaties. Voor elke recursieve functieaanroep is een aantal klokcycli nodig en ook moet elke keer een aantal registers op de *call stack* geplaatst worden. Een eenvoudige *for-loop* werkt dan sneller en kost minder geheugen.

**Inheritance**

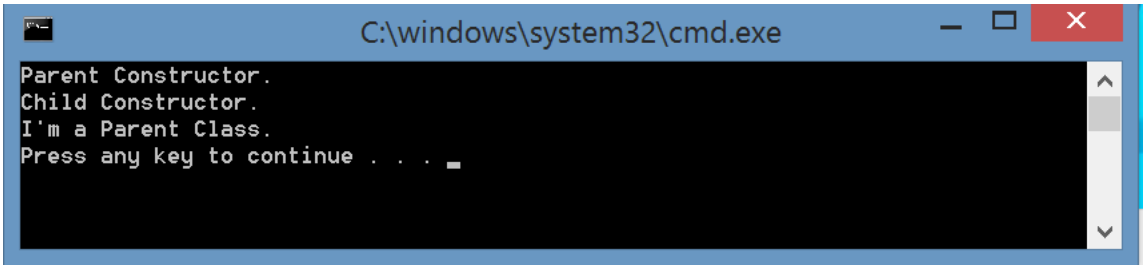
De syntax (schrijfwijze) voor inheritance is:

```
class ParentClass{

...parent class code

}

class ChildClass : ParentClass{
...child class code
}
```

Ofwel overerving in C#. Bekijk de onderstaande broncode:

```
namespace Inheritance
{
    public class ParentClass
    {
        public ParentClass()
        {
            Console.WriteLine("Parent Constructor.");
        }

        public void print()
        {
            Console.WriteLine("I'm a Parent Class.");
        }
    }

    public class ChildClass : ParentClass
    {
        public ChildClass()
        {
            Console.WriteLine("Child Constructor.");
        }

        public static void Main()
        {
            ChildClass child = new ChildClass();

            child.print();
        }
    }
}
```

De uitvoer van dit programma ziet er als volgt uit:




De bovenste klasse is heet ParentClass en de hoofdklasse heet ChildClass. Wat we willen doen is het creëren van een kind klasse, met behulp van bestaande code van ParentClass.

Eerst moeten we onze intentie verklaren om de ParentClass te gebruiken als de basisklasse van ChildClass. Dit wordt bereikt door de ChildClass verklaring **public class ChildClass: ParentClass**. De basis klasse wordt bepaald door het toevoegen van een dubbele punt, ":", na de afgeleide klasse-id en vervolgens met vermelding van de naam basisklasse.

**Opmerking:** C # ondersteunt enkele klasse overerving alleen. Daarom kan slechts één basisklasse gespecificeerd worden om van te erven.

ChildClass heeft precies dezelfde mogelijkheden als ParentClass. Vanwege dit, kun je ook zeggen ChildClass "is" een ParentClass. Dit blijkt uit de methode Main () van ChildClass wanneer de methode print () wordt aangeroepen. ChildClass heeft niet zijn eigen print () methode, dus gebruikt het de methode print () van de ParentClass. De resultaten zijn in de 3e lijn van de output te zien.

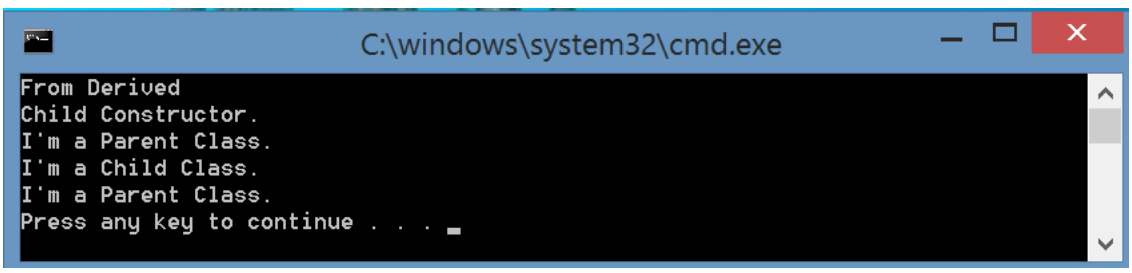
Base klassen worden automatisch geïnstantieerd voor afgeleide klassen. Let op de output. De ParentClass constructeur wordt uitgevoerd voordat de ChildClass constructeur.

# C# Classes en Objects

Bekijk de volgende code:

```
namespace Inheritance2
{
    public class Parent
    {
        string parentString;
        public Parent()
        {
            Console.WriteLine("Parent Constructor.");
        }
        public Parent(string myString)
        {
            parentString = myString;
            Console.WriteLine(parentString);
        }
        public void print()
        {
            Console.WriteLine("I'm a Parent Class.");
        }
    }
    public class Child : Parent
    {
        public Child()
            : base("From Derived")
        {
            Console.WriteLine("Child Constructor.");
        }
        public new void print()
        {
            base.print();
            Console.WriteLine("I'm a Child Class.");
        }
        public static void Main()
        {
            Child child = new Child();
            child.print();
            ((Parent)child).print();
        }
    }
}
```

De uitvoer is:



Afgeleide klassen kunnen met basisklassen communiceren tijdens concretisering. De afbeelding met broncode laat zien hoe dit gedaan wordt. De dubbele punt, ":", en het sleutelwoord **base** roepen de basisklasse constructor aan met de bijpassende parameter lijst. Als de code base ("From Derived") niet was toegevoegd aan de Afgeleide constructor, zou de code automatisch Parent() hebben aangeroepen. De eerste regel van de output laat zien dat de basisklasse constructor wordt aangeroepen met de string "From Derived".

Soms wilt je je eigen implementatie van een methode creëren die bestaat in een basisklasse. De klasse Child doet dit door een eigen print () methode te declareren. De methode Child print () verbergt de methode Parent print (). Het effect is de methode Parent print () niet zal worden aangeroepen, tenzij we iets speciaals doen om ervoor te zorgen dat het wel wordt aangeroepen.

Binnen de methode Child print (), wordt expliciet de methode Parent print () aangeroepen. Dit wordt gedaan door voor de naam van de methode "base." te schrijven. Met behulp van de base sleutelwoord, kunnen alle basisklasse publiek of beschermde klasse leden toegang toe worden verkregen. De uitvoer van de methode Child print () staat in de regels 3 en 4 van de uitvoer.

Een andere manier om toegang te krijgen tot leden van de basisklasse is door middel van een expliciete cast. Dit gebeurt in de laatste instructie van de Child klasse Main (). Vergeet niet dat een afgeleide klasse is een specialisatie van de basisklasse. De laatste regel van de uitvoer toont dat de Parent print () inderdaad uitgevoerd werd.