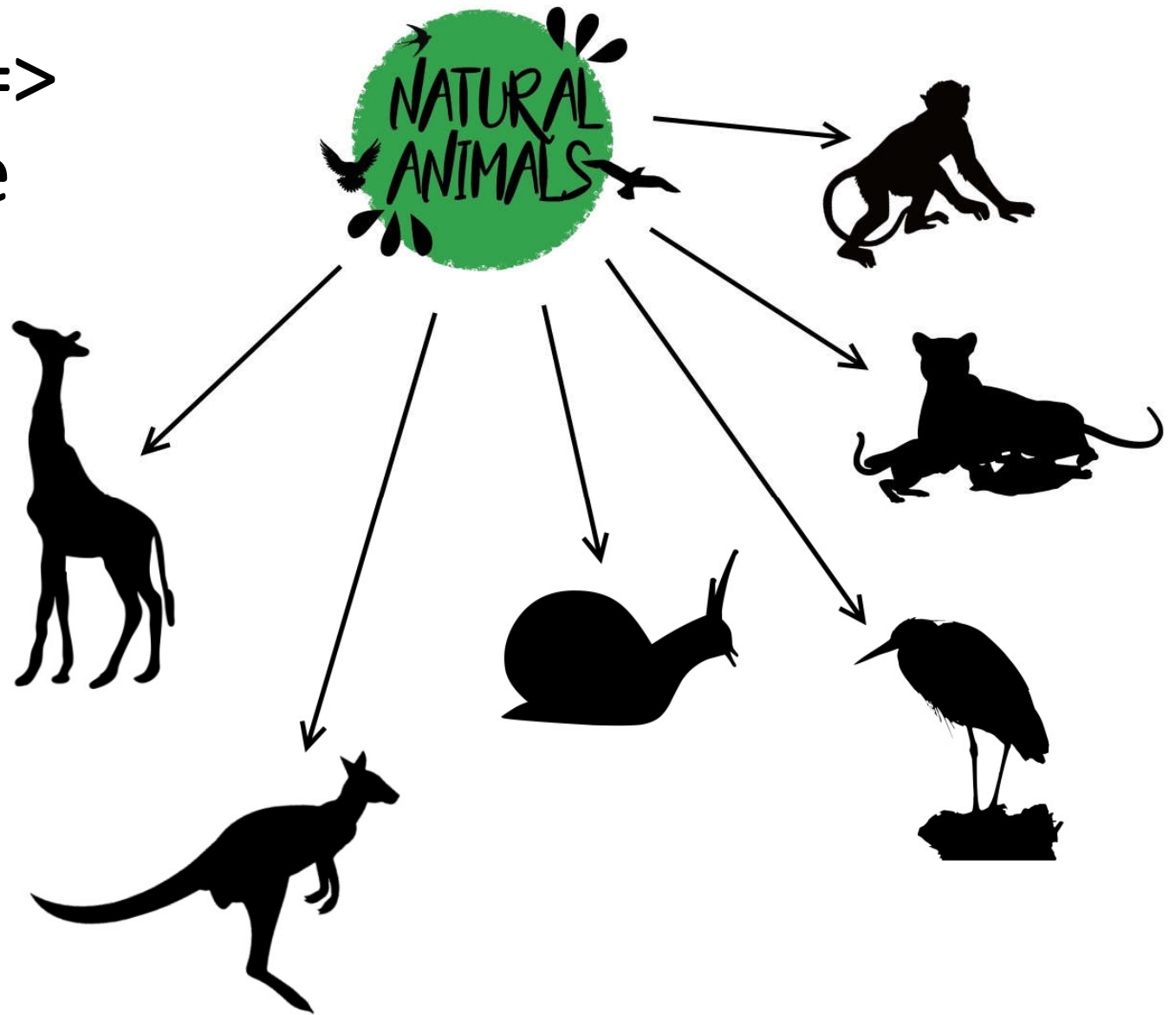


Programmeren in C#

object georiënteerd
programmeren

Overerving =>
Inheritance



Inheritance = overerving

- Een klasse kan 'overerven' van andere klassen om zo hun functionaliteiten uit te breiden. Dit noemt men inheritance

```
public class ParentClass {  
    public void DoSomeWork() {...}  
}  
public class ChildClass : ParentClass {  
}
```

- ChildClass erft dus van ParentClass: ChildClass : **derived** class.
- ChildClass kan beschikken over alles wat de parent heeft en kan daarbovenop nog extra functionaliteiten toevoegen.
- De private members van de parent class kunnen enkel door de parent gebruikt worden.
- De '**protected**' access modifier laat toe dat ook de derived classes die onderdelen kunnen gebruiken.

Upcast en downcast

- Impliciet upcast naar een base class referentie
- Expliciet downcast naar een subclass referentie

```
public class Person {  
    public string lastName;  
    public string firstName;  
    public string address;  
}  
  
public class Client:Person {  
    public string clientID;  
}  
  
static void Demo() {  
    Client clientRef = new Client();  
    Person personRef = clientRef; // Implicit upcast : VALID  
    clientRef = personRef; // Implicit downcast : NOT VALID  
    clientRef = (Client)personRef; // Explicit downcast: VALID  
}
```

Object casting

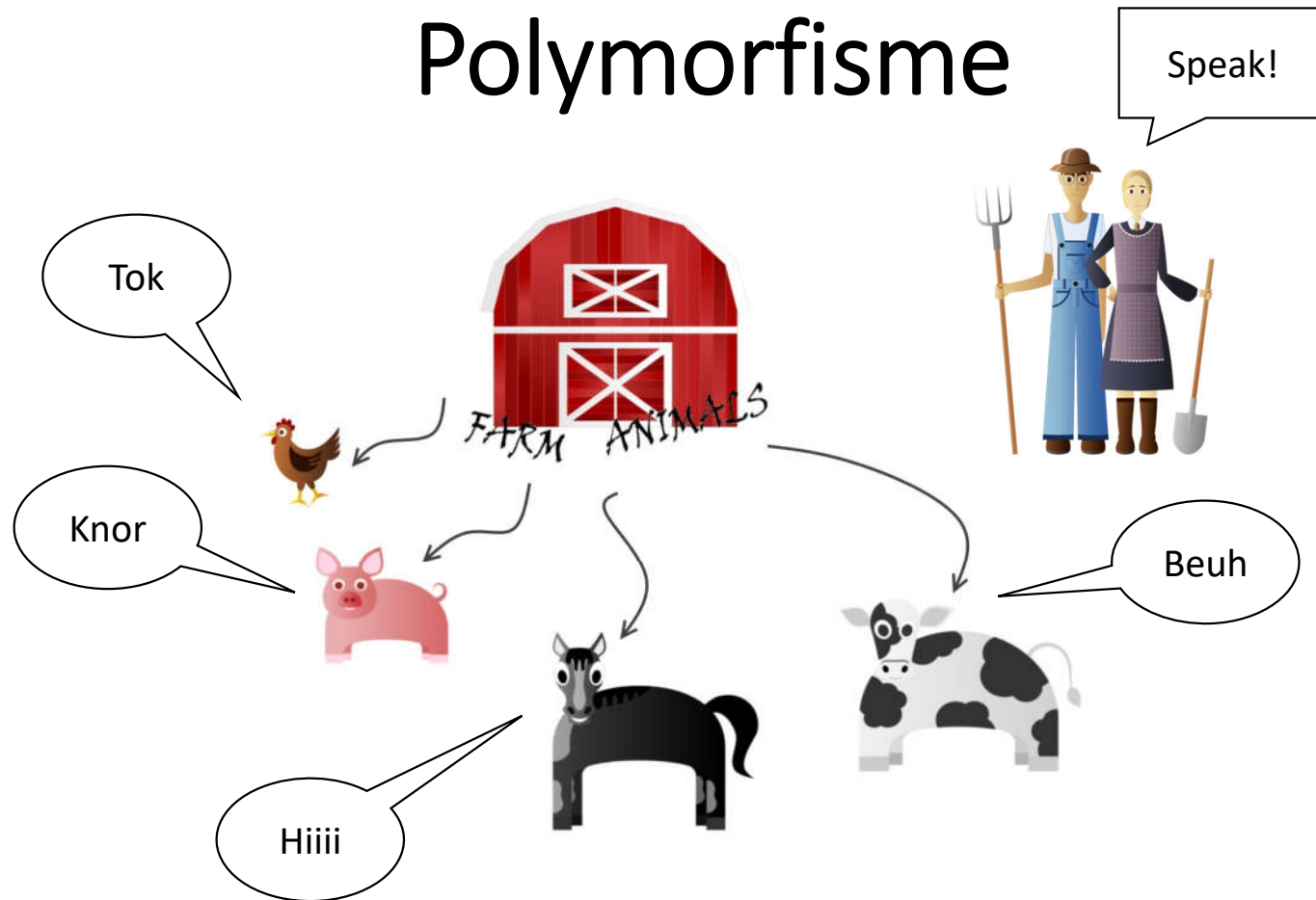
- Indien een expliciete cast faalt, wordt er een `InvalidCastException` geworpen.
- Om veilige cast uit te voeren:
 - We kunnen een expliciete cast uitvoeren met de '`as`' operator. Indien de cast **faalt**, wordt er een **null** reference teruggegeven i.p.v. een exception.
 - Eerst **testen** of het object van het juiste type is overgeërfd met de '`is`' operator.
 - Vanaf C# 7 kunnen we de '`is`' operator combineren met een cast.

```
static void Demo() {  
    Client clientRef;  
    Person personRef = new Person(); // Implicit upcast : VALID  
    Person clientAsPerson = new Client();  
    clientRef = personRef as Client; // clientRef=null  
    clientRef = clientAsPerson as Client; // OK  
} if(personRef is Client) clientRef = (Client)personRef;  
} if(clientAsPerson is Client cl) { Console.WriteLine(cl.clientID); }  
}
```

Labo: overerving of inheritance

- Gebruik de werknemers oefening.
- **Hernoem** werknemersbestand naar **personenbestand**.
- Zorg dat er in dit personenbestand ook **klanten** kunnen opgenomen worden **IN DEZELFDE LIJST!**
- **Enkel werknemers** kunnen **in-dienst** genomen worden, dit moet ook in de klassen duidelijk zijn.
- **Klanten** krijgen een **kortingsschaal** toegekend.
- Maak het mogelijk om naast de volledige lijst ook de **werknemers** en de **klanten op te vragen**.
- We moeten zowel klanten als werknemers kunnen toevoegen.
- Voeg een **functie 'Contacteer'** toe waar we een **boodschap** en onze **naam + titel (optioneel)** kunnen **meegeven** en die een boodschap genereert.

Polymorfisme



Polymorphism of polymorfisme

- Polymorfisme komt van het Grieks en betekent 'veel-vormen'.
- We spreken van 'polymorfisme' als we een gelijkaardige basis hebben voor entiteiten met verschillende implementaties.

```
public class Person {  
    public string lastName;  
    public string firstName;  
    public string address;  
    public string email;  
    public bool sendEmail(){..}  
}  
public class Client:Person {  
    public string clientID;  
}
```


- Client heeft alle kenmerken die Person ook heeft maar voegt er het field clientID aan toe.
- We kunnen Client aanspreken als een Person, maar dan kunnen we het field clientID niet gebruiken.

Het overschrijven van functies

- Het is mogelijk om functies van de 'parent' class te overschrijven met eigen functies die aangepaste functionaliteiten implementeren

```
public class Parent {  
    public bool DoSomeWork() {  
        return true;  
    }  
}  
  
public class Teenager: Parent {  
    public bool DoSomeWork() {  
        return false;  
    }  
}
```

- De functie van Teenager DoSomeWork overschrijft de werking van de Parent class.
- De Compiler zal ook een warning genereren dat de functie de werking van de parent functie overschrijft

 [CS0108](#) 'Werknemer.Contacteer(string, string, string)' hides inherited member 'Persoon.Contacteer(string, string, string)'. Use the new keyword if hiding was intended.

- Om dit te voorkomen dienen we het '**new**' te gebruiken om aan te duiden dat we hiervan bewust zijn. We schrijven dit voor de functie.
 - !! Deze new heeft niets te maken met het keyword '**new**' dat we gebruiken om een instantie van type te maken.

Het overschrijven van functies

- Afhankelijk van het gebruikte type zal de functie gebruikt worden die gekoppeld is aan dit type. Dit noemen we **'early binding'**.

```
public class Parent {  
    public bool DoSomeWork() {  
        return true;  
    }  
}  
  
public class Teenager: Parent {  
    public new bool DoSomeWork() {  
        return false;  
    }  
}
```

- Als we de functie DoSomeWork() aanroepen zal die een ander resultaat geven afhankelijk van het gebruikte type en niet van de oorspronkelijke instantie:

```
Parent ouder = new Parent();  
Teenager tiener = new Teenager();  
Parent tienerVermomdAlsOuder = new Teenager();  
Console.WriteLine(ouder.DoSomeWork() ? "Werken" : "Luieren");  
Console.WriteLine(tiener.DoSomeWork() ? "Werken" : "Luieren");  
Console.WriteLine(tienerVermomdAlsOuder.DoSomeWork() ? "Werken" : "Luieren");
```



Early binding versus late binding

- Om makkelijk met objecten te kunnen werken is het noodzakelijk dat we ervoor kunnen zorgen dat een methode ook wordt aangeroepen vanuit het parent type.
- Hiervoor kunnen we zorgen door in het parent type een methode te decoreren met '**virtual**'.
- De klassen die overerven kunnen een 'virtual' functie overschrijven met het '**override**' keyword.
- We kunnen in een override functie niets wijzigen aan de vorm van de functie!
- Wanneer we werken met 'virtual' en 'override' spreken we van '**late binding**'.

Late binding

- Als we het vorige voorbeeld terugnemen en decoreren de DoSomeWork functie van de Parent klasse met 'virtual' en de functie in de Teenager klasse met override:

```
public class Parent {  
    public virtual bool DoSomeWork() {  
        return true;  
    }  
}  
  
public class Teenager: Parent {  
    public override bool DoSomeWork() {  
        return false;  
    }  
}
```

- We voeren dezelfde bewerking uit als ervoor:

```
Parent ouder = new Parent();  
Teenager tiener = new Teenager();  
Parent tienerVermomdAlsOuder = new Teenager();  
Console.WriteLine(ouder.DoSomeWork() ? "Werken" : "Luieren");  
Console.WriteLine(tiener.DoSomeWork() ? "Werken" : "Luieren");  
Console.WriteLine(tienerVermomdAlsOuder.DoSomeWork() ? "Werken" : "Luieren");
```



Het base keyword

- Het base keyword kan gebruikt worden om overschreven functies van de parent klasse te gebruiken.
- We kunnen het base keyword ook gebruiken in de constructor. Zo kunnen we de constructor van de basisklasse aanroepen in de eigen constructor,

```
public class Parent {  
    protected Parent(decimal myMoney) { Money = myMoney; }  
    public decimal Money { get; private set; }  
    public virtual bool DoSomething() { return true; }  
}  
public class Teenager:Parent {  
    public Teenager(decimal pocketMoney) : base(pocketMoney) { }  
    public override bool DoSomething() { return !base.DoSomething(); }  
}
```

Labo: Late binding 1

- Maak een klasse 'boerderij' aan.
 - Voeg een lijst toe met dieren.
 - Elk dier heeft een aparte klasse met volgende data:
 - Diersoort
 - Naam
 - De functie 'MaakGeluid()'
 - Voeg een functie toe met de naam 'start()'
 - Als de functie wordt aangeroepen maakt elk dier het gepaste geluid
 - (het is voldoende dat dit geluid wordt getoond op het scherm)

Labo: Late binding 2

- Pas het personeelsbestand aan:
 - Maak een aparte klasse aan waarin het adres terechtkomt:
 - De klasse moet volgende data bevatten:
 - Type adres: Thuis, kantoor, buitenverblijf
 - Straat
 - Huisnummer + extra toevoeging
 - Postcode
 - Gemeente
 - Land
 - Overschrijf de ToString methode zodat het adres netjes wordt afgedrukt
 - De functie contacteer(...) moet uitgebreid worden:
 - Adres moet netjes worden afgedrukt
 - De aanspreking moet anders zijn bij de klanten of werknemers
 - Klant: 'Geachte <mijnheer/mevrouw> <naam>
 - Medewerker: 'Beste <naam>
 - Mededeling
 - Aangepaste afsluiting klant/medewerker

Abstracte klassen (abstract classes)

- Van een **abstracte** klasse kan **nooit** direct een **instantie worden gemaakt**. Enkel van de klassen die overerven van deze klasse kunnen instanties worden gemaakt.
- Een abstracte klasse wordt aangemaakt door deze met '**abstract**' te decoreren.
- We kunnen ook **abstracte functies** declareren. Deze functies functioneren als 'virtual' functies maar voorzien **geen implementatie**.
- De implementatie **moet** voorzien worden door de klassen die overerven.

```
public abstract class Parent {  
    public abstract bool DoSomeWork();  
    public bool canDoMoreWork() { return true; }  
}  
public static void Main(string[] args) {  
    Parent p = new Parent();  
}
```

// => Error

Interfaces

- Een interface lijkt op een klasse maar **voorziet** enkel **specificaties** hoe de klasse er moet uitzien en kan **geen implementaties** voorzien.
- Men kan een **interface** dus **vergelijken** met een **abstracte klasse** met **enkel abstracte leden**. We kunnen dus zeggen dat een interface impliciet abstract is.
- Een 'class' kan enkel **1 klasse overerven** maar kan **meerdere interfaces** overerven.
- Een 'struct' kan **niet overerven van een klasse** maar **wel van interfaces**.
- **Members** van een interface zijn **impliciet public** en kunnen **geen 'access modifiers'** declareren.
- Het is gebruikelijk dat men een I voor de naam van een interface zet:
 - Vb: `public interface IMyInterface{ ... }`

Impliciete implementatie van Interfaces

- Impliciete implementatie van een interface:

```
public interface IDemo {  
    bool DoSomething();  
}  
public class Demo:IDemo {  
    public bool DoSomething() {  
        return true;  
    }  
}
```

➤ Bij de impliciete implementatie moet de functie **public** zijn!

- Wanneer er een instantie wordt gemaakt van een object die een interface bevat kan die impliciet gecast worden naar een geïmplementeerd interface type.

```
public static void Main(string[] args) {  
    IDemo myDemo = new Demo();  
}
```

Expliciete implementatie van Interfaces

- We kunnen overerven van meerdere interfaces. Soms is het mogelijk dat verschillende interfaces dezelfde functies bevatten. Dan is een expliciete implementatie nodig:

```
public interface IOmed {  
    int DoSomething();  
}  
public interface IDemo {  
    bool DoSomething();  
}  
public class Demo:IDemo, IOmed {  
    public bool DoSomething() { return true; }  
    int IOmed.DoSomething() { return 0; }  
}
```

- Bij een expliciete implementatie moet de '**public**' decorator **weggelaten** worden
- Omdat er 2 functies gebruikt worden die dezelfde signatuur hebben kan de expliciete declaratie enkel gebruikt worden na een cast!

```
Demo myDemo = new Demo();  
myDemo.DoSomething(); // => impliciete functie uit IDemo (return bool)  
((IOmed)myDemo).DoSomething(); // => expliciete functie uit IOmed (return int)
```

Object type

- Elk type in .Net, zowel 'value types' als 'reference types' zijn overgeërfd van het type '**object**'.
- Het object type bevat enkele functionaliteiten die bijgevolg altijd aanwezig zijn in elk .Net type:
 - ✓ ToString()
Standaard wordt hier de naam van het type teruggegeven in tekst maar we kunnen deze functie overschrijven (late binding) met onze eigen functie en een aangepaste tekst teruggeven.
 - ✓ Equals()
Dit wordt gebruikt om objecten met elkaar te vergelijken.
 - ✓ GetType()
Hiermee is het mogelijk om .Net informatie op te vragen over het gebruikte type.

Labo: Interfaces

- Herwerk de oefening van vorige les 'Recept':
- Het programma moet dezelfde functionaliteiten hebben als de vorige oefening maar gebruik een interface in plaats van delegates.



Boxing en Unboxing

- Omdat elk type, ook value types, overgeërfd zijn van Object kunnen we dit gebruiken om een value type om te zetten naar een reference type. Dit noemen we **boxing**:

```
int num = 3;      // value type
object x = num;   // reference type
```

- Indien we later dit object terug willen overzetten naar zijn oorspronkelijk type doen we dit met een cast. Dit noemen we **unboxing**:

```
object x = 3;      // box int into a reference type
int num = (int)x;  //unbox the reference type into a value type
```

Nullable types

- Als een value type niet bestaat dan is het standaard niet mogelijk om dit type een ongedefinieerd waarde te geven.
- In reference types kunnen we de waarde null toekennen.
- We kunnen dit bewerkstelligen door de waarden te boxen en te unboxen. Dit heeft als nadeel dat het werken met deze variabelen zeer omslachtig wordt.
- Daarom kunnen we in C# een value type '**nullable**' maken door er een **?** voor te plaatsen:

```
int? a = null; // Undefined  
if(a==null) a = 3;
```


Werken met nullable types

- Conversie:

- van value naar reference => impliciet
- van reference naar value => expliciet

```
int? a = null;  
a = 6;  
int b = (int)a;
```

!! Maar opgepast als de nullable variabele nog null blijkt te zijn dan krijgen we een runtime exception !

```
a = null;  
b = (int)a; //System.InvalidOperationException
```

- Om te controleren of het type een waarde bevat kunnen we de **HasValue** property gebruiken

```
b = a.HasValue ? (int)a : 0; //OK
```

Generische types of Generics

- Een generisch type is eigenlijk een voorlopige notatie die bij het declareren wordt vervangen door het echt gebruikte type.
- Meestal aangeduid door de letter T en wordt geplaatst binnen groter en kleiner dan tekens: <T>

```
public class ArrayList<T> {  
    T[] _items;  
    int _currentPos = 0;  
    public ArrayList(int capacity=0) {  
        _items = new T[capacity > 0 ? capacity : 1];  
    }  
    public void Add(T data) {  
        EnsureCapacity(_currentPos);  
        _items[_currentPos] = data;  
    }  
    private void EnsureCapacity(int currentPos) { ... }  
}
```

- Bij de declaratie wordt het type T vervangen met het echte type. We spreken dan over '**open type**' (= <T>) en '**closed type**' (= <int>, <string>...)
 - ArrayList<string> myList = new ArrayList<string>();

Gebruik van 'generics' in functies

- Het is ook mogelijk om generische parameters te gebruiken in functies.
- We declareren net zoals bij een klasse de generics na de declaratie van de functie en kunnen dan generische parameters declareren:

```
public static void Verwissel<T>(ref T v1, ref T v2) {  
    T varRef = v1;  
    v1 = v2;  
    v2 = varRef;  
}
```

...

```
int a = 1; int b = 2;  
Verwissel<int>(ref a, ref b);  
Console.WriteLine($"a = {a} en b = {b}");
```

- Enkel functies kunnen generische types declareren, fields, constructors, operators, events,... kunnen dit niet maar ze kunnen generische types die op klas niveau zijn gedeclareerd wel gebruiken!

```
public T CurrentData { get { return _items[_currentPos]; } }
```

Generische types gebruiken

- Klassen en functies kunnen meerdere generische types declareren.
 - Functies die meerdere generische types overloaden hebben dan ook een andere vorm dan functies met dezelfde naam die minder of geen types declareren

```
public class Dictionary<TKey, TValue>
```

- Default value van een generisch type kan niet null zijn aangezien we niet weten of het een value type is of een reference type.
 - Daarom gebruiken we de default() keyword

```
T _item = default(T); // int = 0, ref types = null
```

- We kunnen beperkingen opleggen aan het generisch type met het keyword where. Dit noemen we generic constraints.
 - Constraints: base-class, interface, class, struct, new()

```
public class ArrayList<T> where T : class {...}
```

Generische delegate

- Het is ook mogelijk om generische parameters te gebruiken in delegates.
- Dit geeft de mogelijkheid om een delegate functie te maken die kan gelden voor elk type.

```
public delegate T IsGreatherThen<T>(T t1,T t2);
public class ArrayList<T> where T : class {
    T[] _items = null;
    int _currentPos = 0;
    public ArrayList(int capacity=0) {
        _items = new T[capacity > 0 ? capacity : 1];
    }
    public void Sort(IsGreatherThen<T> sortFunction) { ... }
    ...
}
...
public static string CompareList(string t1,string t2) {
    return string.Compare(t1,t2) <0 ? t1 : t2;
}
...
ArrayList<string> myList = new ArrayList<string>();
myList.Sort(CompareList);
```

'Func' en 'Action' delegates

- In de .Net System namespace zijn een set van generische delegates gespecificeerd die bijna elke vorm van functies kunnen aannemen.

```
public delegate void Action<in T>(T obj);  
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);  
...  
public delegate TResult Func<in T, out TResult>(T arg);  
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);  
...
```

- Deze delegates zijn beperkt doordat ze **geen ref of out parameters** aanvaarden.
- Omdat we generische functies kunnen overloaden kunnen Func en Action gebruikt worden tot 16 parameters.
- De in- en outkeywords kunnen gebruikt worden om aan te duiden dat het enkel gaat om input of output generics. Dit noemen we **parameter variance**

Gebruik van 'Func' en 'Action'

- Func en Action delegates zijn een standaard oplossing en kunnen vaak gebruikt worden ipv eigen delegates te declareren.

```
public static string SelectItemTest(string s) { return s[0] <= 'g' ? s : "---"; }  
Public static void Main (string[] args){  
    string[] items = { "dierenvoer", "groenten", "zout", "eieren", "spek" };  
    IEnumerable<string> result = items.Select(SelectItemTest);  
}
```

- Ze worden ook gebruikt door de nieuwere functies van het framework (na de introductie van Generics) zoals Linq en in Lambda expressies.

```
Public static void Main (string[] args){  
    string[] items = { "dierenvoer", "groenten", "zout", "eieren", "spek" };  
    IEnumerable<string> result = items.Select(  
        (s,outStr) => { return s[0] <= 'g' ? s : "---";  
    });  
}
```

Extension methods

- Met extension methods kunnen we bestaande types uitbreiden met nieuwe functies zonder het originele type te wijzigen.
- Opbouw van een extension method:
 - Een extension method moet altijd een **statische functie** zijn.
 - Het moet ook **gedeclareerd** worden in een **statische klasse**.
 - De eerste parameter moet het type zijn dat wordt uitgebreid en met het keyword **this** gedecoreerd worden
 - `public static class MyExtensions {`
 - `public static bool NotNullOrEmpty(this string str) {`
 - `return !string.IsNullOrEmpty(str);`
 - `}`
 - `}`
 - ...
 - `Console.WriteLine("Lege string".NotNullOrEmpty() ? "Vol" : "Leeg");`

Labo

1. Maak een Utils of Gereedschap klas aan waarin je volgende extension methods maakt:
 - Controleren of een array niet null of leeg is.
 - Controleert of een string niet null of leeg is.
 - Geef beide functies dezelfde naam
 - Die ervoor zorgt dat de eerste letter van een string een hoofdletter is.
2. Maak een CLI functie aan waarin je tracht een int om te zetten naar een string.
 - Gebruik hiervoor enkel de **Parse** functie.
 - Gebruik ook een try – catch blok en geef de boodschap van de Exception weer op het scherm wanneer de parse faalt samen met de relevante informatie zoals stack trace.

