

leren. durven. doen.



# *C# FUNDAMENTALS*

C# PROGRAMMING

# Inhoud

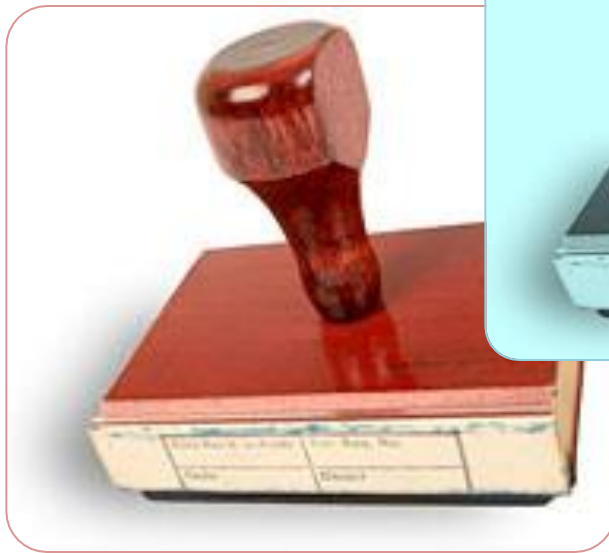
## **GENERICICS**

1. Wat zijn Generics?
2. Custom Generic Class
3. Generic Lists



# *Generic Classes*

## Parametrisering van Classes



# Wat is een Generic Class?

**Via generics kunnen classes met een generiek type T gedefinieerd worden**

De class kan dan geïntantieerd worden met verschillende types

Voorbeelden:

`list<T> → list<int> / list<string> / list<student>`

**hoofddoelstellingen van generieke types:**

1. Verbetering van type safety
2. Vermijden van type castings
3. Elimeneren van box'ing en unbox'ing

Analoog aan C++ templates

# Generic Syntax: voorbeelden Built-in Generics

## Voorbeeld built-in generic class `stack<T> { ... }`

T is the type variable

```
Stack<int> mystack = new stack<int>();
```

## Compiler controleert op type safety

```
Mystack.Push(4.3); // compiler error (geen int type)
```

## Voorbeeld built-in generic class met 2 generic input-types:

```
Dictionary<Tkey, Tvalue>
```

# Terminologie

## Waarom term generic?

Via generieke class kunnen we gedrag/acties (methods) definiëren die onafhankelijk van het type zijn (hergebruik van code).

## Wordt som 'parametric polymorphism' genoemd

We geven een type parameter mee en dezelfde code of gedrag wordt toegepast op dit type.

# Generic parametrization

Generics kunnen gebruikt worden bij:

- Types
  - Struct
  - Interface
  - Class
  - Delegate
- Methods

## Afgeleide classe van generic class

Het is gemakkelijk om een afgeleide class te maken van een generieke class bv:

```
public class IntStack : Stack<int>
{
    ...
}
```



# Waar kunnen Generics gebruikt worden?

Generics kunnen ook gebruikt worden in fields, properties and methods,... van een class:

```
public struct Customer<T>
{
    private static List<T> customerList;
    private T customerInfo;
    public T CustomerInfo { get; set; }
    public int CompareCustomers( T customerInfo );
}
```

## Voorbeeld gebruik van generic class

Een generieke class kan geïntantieerd worden, zoals gewone class, maar met het specifiek generiek type tussen<>


Voorbeeld:

```
Customer<int> fred = new Customer<int>();  
fred.CustomerInfo = 4;
```

# Default value instellen voor type parameter

Bv: wanneer je een variabele van generiek type T moet initialiseren:

```
public class GraphNode<T> {  
    private T nodeLabel;  
    private void ClearLabel() {  
        nodeLabel = null;  
    }  
}
```



Stel: T is  
van type  
**int**?

Waarom werkt bovenstaande code niet?

# Default value instellen voor type parameter

Gebruik hiervoor **default** keyword; bv:

```
public class GraphNode<T> {  
    private T nodeLabel;  
    private void ClearLabel() {  
        nodeLabel = default;  
    }  
}
```

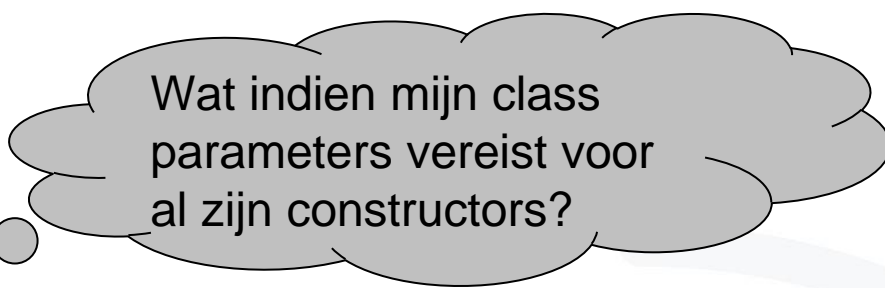
Indien T een reference-type is, zal **default** null zijn,

Voor value-types, is **default** 0 voor numerieke types (alle bits op 0 gezet)

# Generic Constraints

Wat indien we dit willen:

```
public class Stack<T>
    public T PopEmpty() {
        return new T();
    }
}
```



Wat indien mijn class  
parameters vereist voor  
al zijn constructors?

Waarom zal de compiler een foutmelding  
geven op de code hierboven?

# Generic Constraints - class Myclass<T>: where ...

Het keyword **where** specificeert beperkingen (constraints) voor een type parameter T.

Een constraint kan de beperking geven dat T afgeleid is van een andere class, of een interface implementeert bv T moet de interface IDrawable implementeren:


```
public interface IDrawable { public void Draw(); }
```

Bv constraint: T implements the IDrawable interface.

```
public class SceneGraph<T> where T : IDrawable {  
    public void Render() { ... T node; ...  
        node.Draw();  
    }  
}
```

Geen casting nodig

Compiler gebruikt type information van T



Dit kan  
afgedwongen  
worden 'at  
compile time'

# Generic constraint: T moet zelf class zijn

Voorbeeld: class constraint:

```
public class CarFactory<T> where T : class
{
    private T currentCar = null;
    ....
}
```

Zorgt dat bv CarFactory<int> en andere value-types verboden zijn.

In voorbeeld hierboven omdat: int niet gelijk kan worden gesteld aan null.

# Generic constraint: T moet struct zijn

Constraint is data T een a value (struct) type moet zijn, bv:

```
public struct Nullable<T> where T : struct
{
    private T value;
    ....
}
```

Voorbeeld:

```
public class Stack<T> where T : struct {
    public T PopEmpty() {
        return new T();
    }
}
```



# Constructor constraint: where T: new()

- constraint met *new* keyword, bv:

```
public class Stack<T> where T : new()  
{  
    public T PopEmpty() {  
        return new T();  
    }  
}
```

= Parameter-less **constructor constraint**

Type T moet een public parameter-less constructor bezitten.

**Opmerking:** new() constraint moet als laatste constraint worden gespecificeerd indien er meerdere constraints op T zijn.

# Uitgebreide Syntax van Generic Class declaratie

```
class MyClass<type-parameter-list> : class-base
where <type-parameter-constraints-clauses>
{
    // Class body
}
```

## Voorbeeld:

```
class MyClass<T> : BaseClass where T : new()
{
    // Class body
}
```

# Generic Constraints Syntax

```
public SomeGenericClass<some parameters>  
    where type-parameter :  
        primary-constraint,  
        secondary-constraints,  
        constructor-constraint
```

```
public class MyClass<T>  
    where T : class, IEnumerable<T>, new()  
    {...}
```

# Generic Constraints

## Primary constraint:

**class** (reference type parameters)

**struct** (value type parameters)

## Secondary constraints:

Interface derivation

Base class derivation

## Constructor constraint

**new()** – parameterloze constructor constraint

# Generic Constraints – Primary constraint

Een generic type parameter, kan geen, één of meerdere primary constraints hebben, waaronder:

- Moet afgeleid zijn van een non-sealed concrete of abstracte base type
- class constraint
- struct constraint

# Generic Constraints – Secondary constraint

Een generic type parameter, kan geen, één of meerdere interface constraints hebben,

Voorbeeld:

```
public class GraphNode<T> {  
    where T : ICloneable, IComparable  
    ...  
}
```

# Generic Constraints – Beperking where clause

Een type parameter kan maar één enkele **where** clause bezitten, alle constraints moeten worden gespecificeerd binnen één where clause.

Niet mogelijk:

```
public class GraphNode<T> {  
    where T : MyNode, ICloneable  
    where T : IComparable, New()  
    ...  
}
```

# Generic Constraints – Meerdere Type parameters

Een generic type kan geparameteriseerd worden met meerdere type place-holders, voorbeeld:

```
public interface IFunction<TDomain,TRange> {  
    TRange Evaluate(TDomain sample);  
}
```

Generiek type voor bv 2D, 3D, complex function support



# Generic Constraints – Afhankelijkheden tussen generieke type parameters

Elke type parameter kan zijn eigen constraints hebben, en dus zijn eigen **where** clause.

Het is mogelijk om een type parameter te specificeren die afhankelijk is van een andere type parameter, Voorbeeld:

```
public class SubSet<U,V> where U : V
```

```
public class Group<U,V>
```

```
    where V : IEnumerable<U> { ... }
```

# Generic Constraints – Compile errors

- `class A { ... }`
- `class B { ... }`
- `class Incompat<S, T>`  
    `where S: A, T`  
    `where T: B {`  
        `... }`
- `class StructWithClass<S, T, U>`  
    `where S: struct, T`  
    `where T: U`  
    `where U: A { ... }`

# Generic Constraints – Compilatie errors

```
interface I<T>
{
    void F();
}

class X<U,V>: I<U>, I<V>
{
    void I<U>.F() { ... }
    void I<V>.F() { ... }
}
```

# *Generic Lists*



# Generic Lists?

**Via generics kunnen classes met een generiek type T gedefinieerd worden**

**De class kan dan geïntantieerd worden met verschillende types**

**Voorbeelden:**

**`list<T> → list<int> / list<string> / list<student>`**

**Generics worden ook "parameterized types" of "template types" genoemd**

**Analoog aan de templates in C++**

**Analoog aan de generics in java**

# Generics – Voorbeeld

T is een ongekend type, parameter van de class

```
public class GenericList<T>
{
    public void Add(T element) {}
```

T kan gebruikt worden binnen elke method in de class

```
static void Main()
{
```

```
    // Declare a list of type int
    GenericList<int> intList =
        new GenericList<int>();
    // Declare a list of type string
    GenericList<string> stringList =
        new GenericList<string>();
}
```

T kan bv bij creatie van object (instantiëring) vervangen worden door **int**

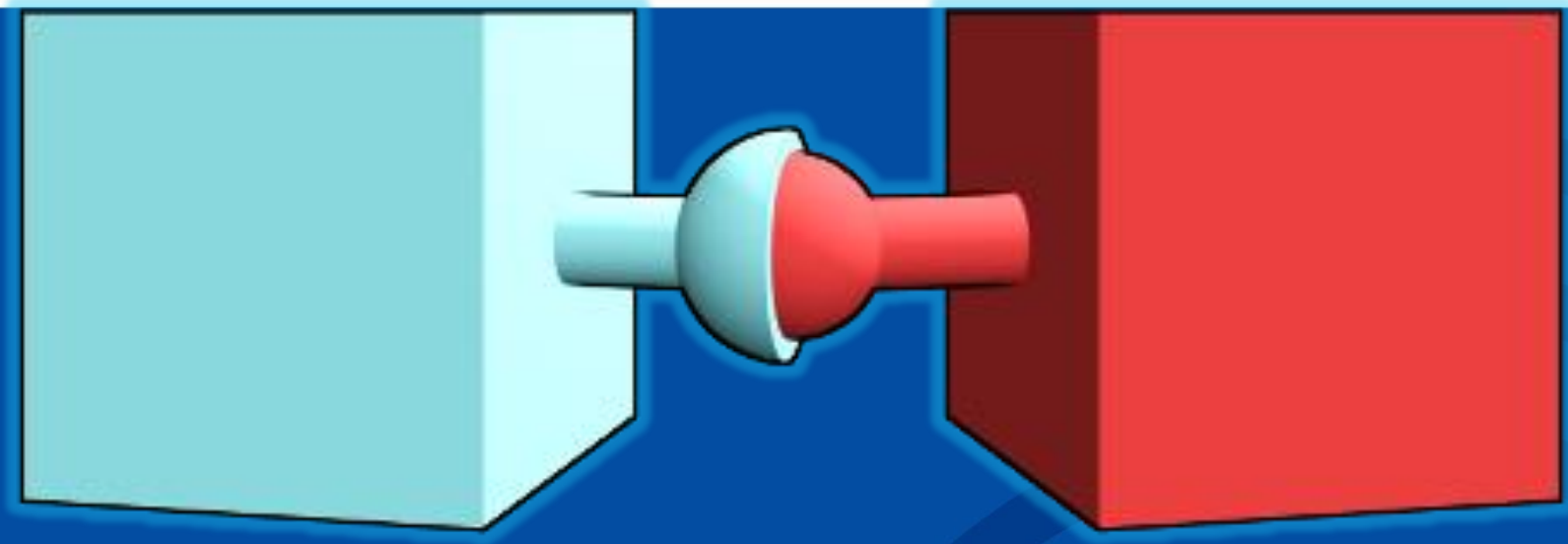
# GenericList<T>

```
public class GenericList<T>
{
    void Add(T element) { ... }
}

class GenericListExample
{
    static void Main()
    {
        // Declare a list of type int
        GenericList<int> intList =
            new GenericList<int>();
        // Declare a list of type string
        GenericList<string> stringList =
            new GenericList<string>();
    }
}
```

## Demo

leren. durven. doen.



# *Generic Constraints*

DEMO



# Generic Method – Voorbeeld

```
public static T Min<T>(T first, T second)
    where T : IComparable<T>
{
    if (first.CompareTo(second) <= 0)
        return first;
    else
        return second;
}

static void Main()
{
    int i = 5;
    int j = 7;
    int min = Min<int>(i, j);
}
```

leren. durven. doen.



*Vragen?*

## REFERENTIES

PRO C# 7 WITH .NET AND .NET CORE – ANDREW  
TROELSEN – PHILIP JAPIKSE

[HTTPS://WWW.LEARNCS.ORG/](https://www.learncs.org/)

FUNDAMENTALS OF COMPUTER PROGRAMMING WITH C#

© SVETLIN NAKOV & CO., 2013