

Generics

Mere genbrug

OOP2020

Thomas Bøgholm

Statisk typesikkerhed, Typer i C#

Motivation for generics

- C# er et statisk typet sprog
 - **Vi kan afgøre typer på compiletime!**
 - Konsekvenser
 - Færre fejl foresaget af typer (færre runtimefejl)
 - Hjælpssom compiler (fejl rapporteres tidligt)
 - Code completion
 - Sikker refaktorering
 - **Streng compiler (bliver sur hvis du ikke overholder reglerne)**
 - **kode kræver ofte gennemtænkt design**
 - Er det skidt?
 - Svært at lave 'bad hacks'

Alternativt, ville vi først vide
det når programmet kører!

Eksempel

- C#s typeregler siger:
 - Man må ikke blande **pærer** og **bananer**
- Dette kan afgøres på **compiletime**, fordi vi altid kender typerne (statisk typet)

```
string s = "tekst";  
int i = 500;  
s = i;  
i = s;
```

[🔍] (local variable) **int** i

Cannot implicitly convert type 'int' to 'string'

Cannot convert source type 'int' to target type 'string'

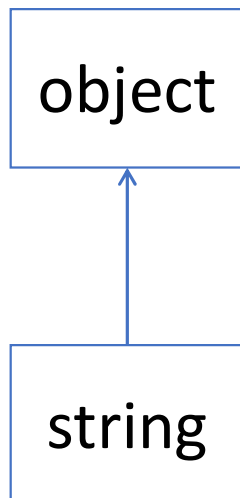
[🔍] (local variable) **string** s

Cannot implicitly convert type 'string' to 'int'

Cannot convert source type 'string' to target type 'int'

Typecasts

- Det er klart at nogle gange ved vi mere(tror vi) end compileren:



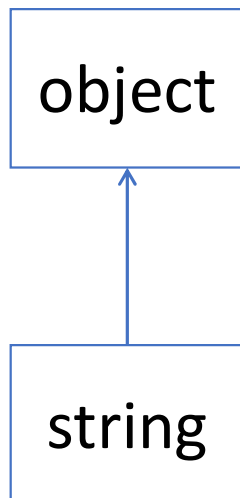
```
string s = "tekst";  
object o = s;
```

```
string finstreng = o;
```

Men det kan jo aldrig gå galt?!?

Typecasts

- Det er klart at nogle gange ved vi mere(tror vi) end compileren:

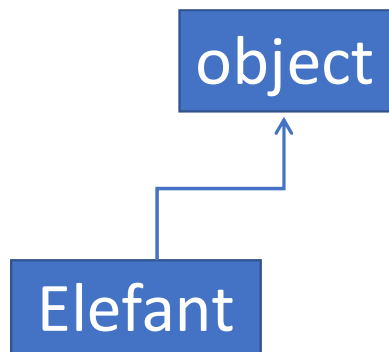


```
string s = "tekst";  
object o = s;
```

```
string finstreng = (string)o;
```

Fix det med et cast:
"Jeg lover jeg ved hvad jeg gør!"

Ikke-trivielt eksempel



```
class Elefant
{
    public int Alder { get; set; }
}
```

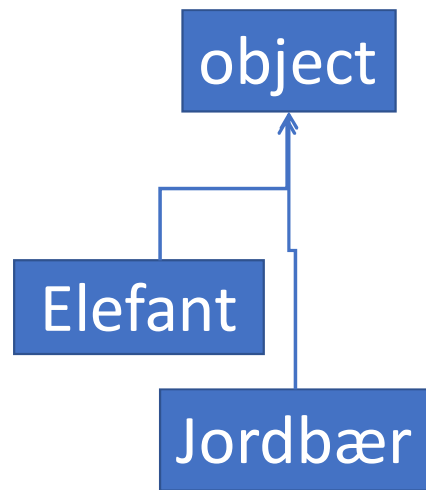
- En generel liste?

```
interface List
{
    object Get(int index);
    void Add(object o);
}
```

```
list.Add(new Elefant());
int alder = list.Get(0).Alder
```

List.Get returnerer object!
- object har ikke alder
TypeCast?

Ikke-trivielt eksempel



- En generel liste?

```
class Elefant
{
    public int Alder { get; set; }
}
```

```
interface List
{
    object Get(int index);
    void Add(object o);
}
```

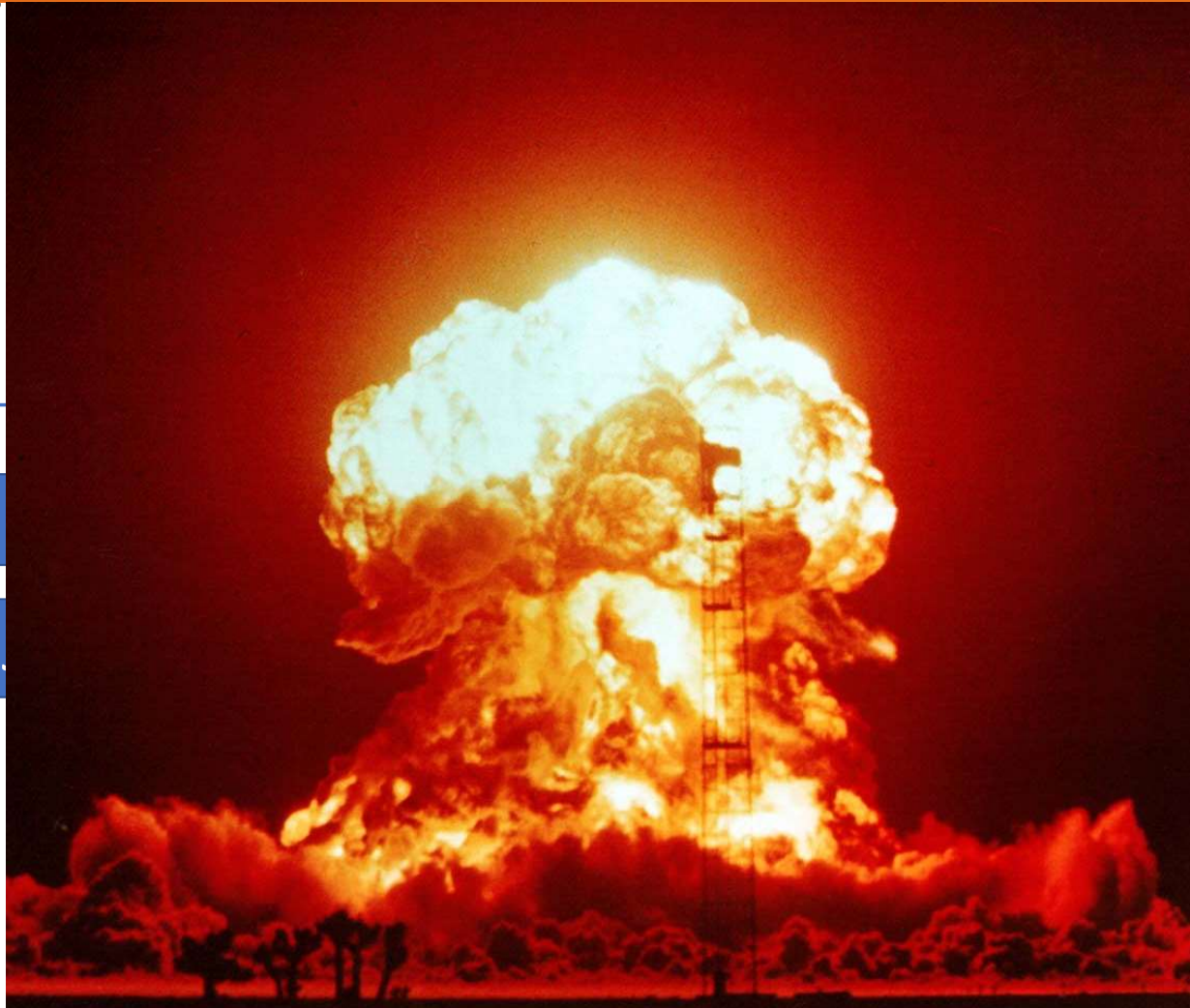
```
list.Add(new Elefant());
int alder = ((Elefant)list.Get(0)).Alder
```

Hvad hvis nogen tilføjede Jordbær!?

Hvad hvis nogen tilføjede Jordbær!?

Ikke-tri

Elefa



);

Generics

- Generiske typer fixer det hele!
 - **Vi kan lave typer der er parametriserede!**
 - Klasseskabelon med typeparameter (placeholder type)
- `List<int>`, `List<string>`, `List<Person>`, ...
 - `Stack<int>`, `Stack<double>`, `Stack<Vehicle>`,...

Genbrug **uden** generiske typer

- Før generiske typer (*System.Collections*) indeholdt collection klasser elementer af typen **System.Object**.
- Alle objekter arver fra System.Object -> collection klasser kunne genbruges til forskellige typer
- **Problemer:**
 - Ingen compile-time type-sikkerhed:
 - Data ud -> Eksplicit typekonvertering (cast) påkrævet
 - mulig exception på kørselstidspunktet + mudret kode

Option 1: generel stak der kan holde elementer af enhver type

```
public class ObjectStack
{
    private int position;
    private object[] data;
    public ObjectStack(int length) { data = new object[length]; }
    public void Push(object obj) { data[position++] = obj; }
    public object Pop() { return data[--position]; }
}
```

```
static void Main(string[] args)
{
    ObjectStack os = new ObjectStack(10);
    os.Push("hello");
    os.Push(7);
    os.Push(new Person("Kim"));
    int a = (int)os.Pop(); //Cast - og runtime fejl.
}
```

Option 2: IntStack, PersonStack, StringStack, ... giver statisk typesikkerhed, men smertefuldt og (potentielt) fejlbehæftet at implementere.

Option 1: generel stak der kan holde elementer af enhver type

```
public class ObjectStack
{
    private int position;
    private object[] data;
    public ObjectStack(int length) { data = new object[length]; }
    public void Push(object obj) { data[position++] = obj; }
    public object Pop() { return data[--position]; }
}
```

```
static void Main(string[] args)
{
    ObjectStack os = new ObjectStack(10);
    os.Push("hello");
    os.Push(7);
    os.Push(new Person("Kim"));
    int a = (int)os.Pop(); //Cast - og runtime fejl.
}
```

Option 2: IntStack, PersonStack, StringStack, ... giver statisk typesikkerhed, men smertefuldt og (potentielt) fejlbehæftet at implementere.

```
class ElephantList : IEnumerable
{
    private IList elephants;
    IEnumerator IEnumerable.GetEnumerator()
    {
        return elephants.GetEnumerator();
    }
    public void Add(Elephant item)
    {
        elephants.Add(item);
    }
    public void Clear()
    {
        elephants.Clear();
    }
    public bool Contains(Elephant item)
    {
        return elephants.Contains(item);
    }
    public void CopyTo(Elephant[] array, int arrayIndex)
    {
        elephants.CopyTo(array, arrayIndex);
    }
    public void Remove(Elephant item)
    {
        elephants.Remove(item);
    }
    public int Count => elephants.Count;
    public bool IsReadOnly => elephants.IsReadOnly;
    public int IndexOf(Elephant item)
    {
        return elephants.IndexOf(item);
    }
    public void Insert(int index, Elephant item)
    {
        elephants.Insert(index, item);
    }
    public void RemoveAt(int index){
        elephants.RemoveAt(index);
    }
    public Elephant this[int index]{
        get => (Elephant)elephants[index];
        set => elephants[index] = value;
    }
}
```

Problemer med object-tilgang

- Eneste måde at opnå statisk typesikkerhed er, at implementere specialiseret datastruktur
 - Specifikke implementationer
 - CarList,
 - IntList,
 - StringList,
 - **ElephantList**
- Besværlig/Ingen genbrug

Samme problem for metoder

- ... En metode der var anvendelig for forskellige typer skulle implementeres ved brug af object eller i en udgave for hver type.
 - Ligeså lidt typesikkerhed eller lige så smertefuldt.
- Generics giver løsningen på disse problemer!

Genbrugbar, type-sikker kode

- Generics giver type-sikker genbrugbar kode
 - Ingen casts
- **Type-parametre** (placeholder typer) erstattes med faktiske **type-argumenter** af klienter.
- Type-parameter angives <T>
- **T** er type-parameter

Generisk stak

```
public class Stack<T>
{
    private int position;
    private T[] data;
    public Stack(int length) { data = new T[length]; }
    public void Push(T obj) { data[position++] = obj; }
    public T Pop() { return data[--position]; }
}
```

Type-parameter

```
static void Main(string[] args)
{
    Stack<int> intStack = new Stack<int>(10);
    //intStack.Push("hello"); //forhindres på compile-time
    intStack.Push(6); //OK
    //string s = intStack.Pop() //forhindres på compile-time
    Stack<string> stringStack = new Stack<string>(10);
    Stack<Person> personStack = new Stack<Person>(10);
}
```

Type-argument

Hvad der sker med stakken:

```
public class Stack<int> ← Med int som type-argument
{
    private int position;
    private int[] data;
    public Stack(int length) { int[] data = new int[length]; }
    public void Push(int obj) { data[position++] = obj; }
    public int Pop() { return data[--position]; }
}
```

```
public class Stack<string> ← Med string som type-argument
{
    private int position;
    private string[] data;
    public Stack(int length) { string[] data = new string[length]; }
    public void Push(string obj) { data[position++] = obj; }
    public string Pop() { return data[--position]; }
}
```

Og vi kunne lave en generisk liste, graf, dictionary, ...

Flere type-parametre

- Generiske typer kan have flere type-parametre.
- Forskelligt antal/type af type-argumenter angiver forskellige typer

```
public class MyTuple<TItem1, TItem2> //Tuple-definition
{
    public TItem1 Item1 { get; set; }
    public TItem2 Item2 { get; set; }
}
```

```
public static void TupleTest()
{
    MyTuple<int, string> turingEvent = new MyTuple<int, string>();
    turingEvent.Item1 = 1912;
    turingEvent.Item2 = "Allan Turing was born";
    string turingName = turingEvent.Item2;

    MyTuple<int, string> turingEvent1 = turingEvent; //OK - samme type
    MyTuple<string, int> crazytuple = turingEvent; //!!! - forkerte typer!
}
```

Nedarvning af generiske typer

- En generisk klasse kan nedarves ligesom ikke-generisk klasse.
- Subklassen kan lade superklassens type-parametre være *åbne* eller den kan *lukke* dem.

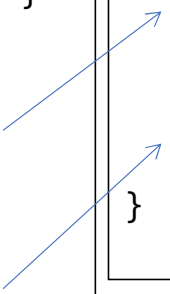
```
class SuperClass<T>
{
    public T Element { get; set; }
}

//åben
class OpenSub<U> : SuperClass<U>
{ }

//lukket
class StringSub : SuperClass<string>
{ }
```

```
void Bar()
{
    OpenSub<int> oSub = new OpenSub<int>();
    oSub.Element = 10;

    StringSub sSub = new StringSub();
    sSub.Element = "Super";
}
```



Nedarvning af generiske typer med nye typeparametre

- Subklasse kan også introducere yderligere type-parametre

<pre>class GBase<T> { public T Element { get; set; } } //ny type-parameter + lukning class Sub1<T> : GBase<string> { public T Element2 { get; set; } } //ny type-parameter; åben class Sub2<T1, T2> : GBase<T1> { public T2 Element2 { get; set; } }</pre>	<pre>public void Foo() { Sub1<double> s1 = new Sub1<double>(); s1.Element = "go"; s1.Element2 = 4.76; Sub2<bool,int> s2= new Sub2<bool, int>(); s2.Element = true; s2.Element2 = 4.76; }</pre>
--	---

Indlejrede generiske typer

- type-parametre fra ydre type bruges er også tilgængelige i indlejrede typer
- Genbrug af type-parameter navn i indlejret type skjuler den ydre classes type-parameter (og giver compiler advarsel)

```
public class Container<T, U> {  
    public class Nested<U> { // Compileradvarsel.  
        public void Foo(T param0, U param1) {}  
    }  
}
```

```
void Bar()  
{  
    Container<int, double>.Nested<string> n;  
  
    n = new Container<int, double>.Nested<string>(); //lovligt  
    n.Foo(42, "Det er SVARET"); //U er lukket med string  
}
```

Selv-referende generiske typer

- En type kan angive sig selv som type-argument når type-parameter lukkes.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

```
class Person : IEquatable<Person>
{
    public readonly string Cpr;
    public Person(string cpr) { this.Cpr = cpr; }

    public bool Equals(Person p)
    {
        return this.Cpr == p.Cpr;
    }
}
```

```
void Foo()
{
    Person p1 = new Person("010163-1919");
    Person p2 = new Person("010163-1919");
    Console.WriteLine(p1.Equals(p2)); //true
}
```

Statisk data og generiske typer

- Statisk data er unik for hver lukket type.

```
class GC<T>
{
    public static int K;
}
public void Foo()
{
    WriteLine(++GC<int>.K);    //1
    WriteLine(++GC<int>.K);    //2


    WriteLine(++GC<string>.K); //1
    WriteLine(++GC<string>.K); //2

    WriteLine(++GC<object>.K); //1
    WriteLine(++GC<object>.K); //2

    WriteLine(++GC<Pear>.K);   //1
    WriteLine(++GC<Pear>.K);   //2

    WriteLine(++GC<Banana>.K); //1
    WriteLine(++GC<Banana>.K); //2
}
```

Increment K



Angivelse af Default værdi

- Forskellige typer har forskellige default værdier
- **default** operator giver default værdi for enhver data type.
 - null og 0 for henholdsvis referencetyper og værdityper(int, float etc.)

```
public class MyTuple<TItem1, TItem2>
{
    public MyTuple()
    {
        this.Item1 = default(TItem1); //default værdi for TItem1
        this.Item2 = default(TItem2); //default værdi for TItem2
    }
    public TItem1 Item1 { get; set; }
    public TItem2 Item2 { get; set; }
}
```


Generiske metoder

- Udover at definere generiske typer kan vi også definere *generiske metoder* for genbrugbare algoritmer.
- En generisk metode erklærer type-parametre sammen med metodens signatur
- Scope: Metode-niveau i stedet for klasse-niveau.
- Generelt anvendelige (generiske) metoder kan med fordel drage gavn af type-parameterisering:

```
public T TypeCast<T>(object o)
{
    return (T)o;
}
```

```
void Foo() {
    object o = new Elephant();
    Elephant elephant = TypeCast<Elephant>(o);
}
```

```

void SetAll<T>(T[] array, T val)
{
    for (int i = 0; i < array.Length; i++)
    {
        array[i] = val;
    }
}

T[] CreateInitializedArray<T>(int size, T initialValue)
{
    var result = new T[size];
    for (int i = 0; i < result.Length; i++)
        result[i] = initialValue;
    return result;
}

void FooBar()
{
    var barray = CreateInitializedArray(10, true);
    var otherarray = CreateInitializedArray(100, 1000M);
    var tarray = CreateInitializedArray(500, "Thomas");

    SetAll(tarray, "Thomas Bøgholm");

    TYPEFEJL! → SetAll(tarray, true);
}

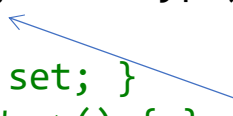
```

Generiske typer og metoder

- Metoder i en generisk type er ikke automatisk generiske.
- Kun metoder der *introducerer* en type-parameter er generiske.
- Properties og constructors kan ikke være generiske (hvorfor?)

```
public class MyClass<T>
{
    void DoSomethingWithVal(T val) { } //ikke-generisk metode
    void DoSomethingElse<T2>(T2 newType) { } //generisk metode

    //T2 Name<TNoWay> { get; set; }
    //public MyClass<TAlsoNoWay>() { }
}
```



Introduction af type-parameter T2

Generiske constraints

- Som udgangspunkt er type-parametre *unconstrained* – alle type-argumenter godtages.
- Vi kan dog bruge **constraints** til at indsnævre lovlige type-args.
- Constraints sikrer at bestemte operatorer el. metoder kan bruges.
- Der findes tre overordnede typer constraints:
 1. *Derivations-constraint*:
 - where** $T : BT$ // T arver fra BT (aka **Base class constraint**)
 - where** $T : IT$ // T implementerer IT (aka **Interface constraint**)
 2. *Reference-type/værdi-type constraint*
 - where** $T : \text{class}$ // T er en reference-type (aka **Class constraint**)
 - where** $T : \text{struct}$ // T er en værdi-type (aka **Struct constraint**)
 3. *Default constructor constraint*
 - where** $T : \text{new}()$ // T har en default (parameterløs) constructor

Eksempel på derivations-constraint

- For at sammenligning af to instanser skal de implementere *IComparable*.
 - Constraint sikrer statisk at type-parametre er *IComparable*
- (*IComparable* -> numeriske typer, strings, chars, *DateTime*, ...).

```
public interface IComparable<T> {    int CompareTo(T other); }
```

constraint

```
T Max<T> (T a, T b) where T : IComparable<T>  
{
```

```
    //interface constraint sikrer at dette er tilladt
```

```
    return a.CompareTo(b) > 0 ? a : b;}
```

```
int x      = Max(21, 4);
```

```
string y   = Max("abe", "kat");
```

```
DateTime z = Max(DateTime.Now, DateTime.Today); ...
```

Eksempel på default constructor constraint

```
void InitializeNulls<T>(List<T> elems) where T : new()
{
    for (int i = 0; i < elems.Count; i++)
        if (elems[i] == null)
            elems[i] = new T();
}
//-----
void Bar()
{
    List<Person> ps = new List<Person>() {
        new Person(), null, null, new Person()
    };
    WriteLine(ps.Count(x => x == null)); //2
    InitializeNulls(ps);
    WriteLine(ps.Count(x => x == null)); //0
}
```

Kræver default constructor for T



Flere constraints

- En type-parameter kan angive multiple constraints (, adskilt)
- Regler ved multiple constraints:
 - Base class constraint skal stå først
 - Default constructor constraint skal stå sidst
- Compiler forhindrer ulovlige kombinationer

```
//T afgrænset til referencetyper med default constructor
class Sample1<T> where T : class, new() { }
```

```
//T afgrænset til værdityper der implementerer IComparable
class Sample2<T> where T : struct, IComparable<T> { }
```

```
class Sample3<T, U>
    where T : class      //T afgrænset til referencetype
    where U : struct, T //U afgrænset til værdityper der
                        //implementerer T
{ }
```

Constraints og nedarvning

- Constraints på klasser skal angives eksplicit i subklasser
 - Giver læsbarhed og velovervejet kode
- Subklasser kan derudover tilføje yderligere constraints
- Constraints på virtuelle metoder arves og kan IKKE ændres.

```
class Sample1<T> where T : class
{
    public virtual void MyCompare<T2>(T2 arg1, T2 arg2)
        where T2 : IComparable<T2>
    { }
}

class Sample2<T> : Sample1<T> where T : class
{
    public override void MyCompare<T2>(T2 arg1, T2 arg2) { }
}
```