C# (C SHARP)

- “Microsofts Antwort auf Java”

C# (C SHARP)

- “Microsofts Antwort auf Java”
- Standardisierung durch die ECMA und ISO

C# (C SHARP)

- “Microsofts Antwort auf Java”
- Standardisierung durch die ECMA und ISO
- im folgenden: Mono

Warum es immer mit Java verglichen wird

- Ähnliche Syntax

Warum es immer mit Java verglichen wird

- Ähnliche Syntax
- Implementierung: Virtuelle Maschine
- Java Virtual Machine $\Leftrightarrow$ Common Language Runtime

Warum es immer mit Java verglichen wird

- Ähnliche Syntax
- Implementierung: Virtuelle Maschine
- Java Virtual Machine $\Leftrightarrow$ Common Language Runtime
  $\rightarrow$ Bytecode $\Leftrightarrow$ Intermediate Language Code

Warum es immer mit Java verglichen wird

- Ähnliche Syntax
- Implementierung: Virtuelle Maschine
- Java Virtual Machine $\Leftrightarrow$ Common Language Runtime
  $\rightarrow$ Bytecode $\Leftrightarrow$ Intermediate Language Code
- Identische Konzepte für

Warum es immer mit Java verglichen wird

- Ähnliche Syntax
- Implementierung: Virtuelle Maschine
- Java Virtual Machine $\Leftrightarrow$ Common Language Runtime
  - $\rightarrow$ Bytecode $\Leftrightarrow$ Intermediate Language Code
- Identische Konzepte für
  - Objekte und Referenzen

Warum es immer mit Java verglichen wird

- Ähnliche Syntax
- Implementierung: Virtuelle Maschine
- Java Virtual Machine $\Leftrightarrow$ Common Language Runtime
  - $\rightarrow$ Bytecode $\Leftrightarrow$ Intermediate Language Code
- Identische Konzepte für
  - Objekte und Referenzen
  - Garbage Collection und Finalization

Warum es immer mit Java verglichen wird

- Ähnliche Syntax
- Implementierung: Virtuelle Maschine
- Java Virtual Machine $\Leftrightarrow$ Common Language Runtime
  - $\rightarrow$ Bytecode $\Leftrightarrow$ Intermediate Language Code
- Identische Konzepte für
  - Objekte und Referenzen
  - Garbage Collection und Finalization
  - Einfach-Vererbung für Objekte, Mehrfach- für Interfaces

Warum es immer mit Java verglichen wird

- Ähnliche Syntax
- Implementierung: Virtuelle Maschine
- Java Virtual Machine $\Leftrightarrow$ Common Language Runtime
  - $\rightarrow$ Bytecode $\Leftrightarrow$ Intermediate Language Code
- Identische Konzepte für
  - Objekte und Referenzen
  - Garbage Collection und Finalization
  - Einfach-Vererbung für Objekte, Mehrfach- für Interfaces
  - Generics

Der wohl wichtigste Unterschied: Unifiziertes Typsystem

Basis-Datentypen sind Objekte:

```
int i=5;
object o=i;
Stack s=new Stack();
s.push(i);
i=(int)s.pop();
```

Der wohl wichtigste Unterschied: Unifiziertes Typsystem

Basis-Datentypen sind Objekte:

```
int i=5;
object o=i;
Stack s=new Stack();
s.push(i);
i=(int)s.pop();
```

Dennoch Unterscheidung von value- und reference-Typen

Der wohl wichtigste Unterschied: Unifiziertes Typsystem

value-Typen: kein Aliasing

```
int i = 5;
int j = i;
j = 6;              // i = 5, j = 6
```

Der wohl wichtigste Unterschied: Unifiziertes Typsystem

reference-Typen: Aliasing

```
class A {
   int f;
}
A a=new A(); a.f=5;
A b=a;
b.f=6;              // a.f = 6, b.f = 6
```

Der wohl wichtigste Unterschied: Unifiziertes Typsystem

reference-Typen: Aliasing

```
class A {
   int f;
}
A a=new A(); a.f=5;
A b=a;
b.f=6;              // a.f = 6, b.f = 6
```

- value-Typen lassen sich in reference-Typen konvertieren
  → *boxing/unboxing*

Der wohl wichtigste Unterschied: Unifiziertes Typsystem

reference-Typen: Aliasing

```
class A {
   int f;
}
A a=new A(); a.f=5;
A b=a;
b.f=6;              // a.f = 6, b.f = 6
```

- value-Typen lassen sich in reference-Typen konvertieren
  → *boxing/unboxing*
- sehr einfach: `int i = 6; object o = i;`

Der wohl wichtigste Unterschied: Unifiziertes Typsystem

reference-Typen: Aliasing

```
class A {
   int f;
}
A a=new A(); a.f=5;
A b=a;
b.f=6;              // a.f = 6, b.f = 6
```

- value-Typen lassen sich in reference-Typen konvertieren
  → *boxing/unboxing*
- sehr einfach: `int i = 6; object o = i;`
- (in Java: `Integer j = new Integer(i);`)

”Hello World” in C#

```
using System;                    //Namespace

public class HelloWorld {
   public static void Main() {
      Console.WriteLine("Hello C# World :-)");
   }
}
```

”Hello World” in C#

```
using System;                    //Namespace

public class HelloWorld {
   public static void Main() {
      Console.WriteLine("Hello C# World :-)");
   }
}
```

Namespace-Konzept von C++, statt package-Konzept von Java

Neue alte Features: Pointer und Pointer-Arithmetik

```
public class Foo {
  public unsafe void Bar() {
    int[] nums = new int[10];
    int* p = &nums[0];

    for (int i = 0;i < 10; i++) {
      *p = 1;
      p++;
    }
  }
}
```

Pointer-Verwendung nur in als unsafe gekennzeichnetem Code

Neue alte Features: Bedingte Kompilation

```
#DEBUG = true
class A {
   static void debug(string text) {
#if DEBUG
      Console.WriteLine(text);
#endif
   }
   static void g() {
      debug("g called");
   }
}
```

Neue alte Features: Bedingte Kompilation

```
#DEBUG = true
class A {
   static void debug(string text) {
#if DEBUG
      Console.WriteLine(text);
#endif
   }
   static void g() {
      debug("g called");
   }
}
```

Allerdings: Keine Ersetzungen oder Macro-Definitionen, keine Ausdrücke als Bedingungen

Neue alte Features: Operator Overloading

```
class A {
   int f;
   static public bool operator ==(A a, A b) {
      return a.f==b.f; }
   static public bool operator !=(A a, A b) {
      return a.f!=b.f; }
}
```

Neue alte Features: Operator Overloading

```
class A {
   int f;
   static public bool operator ==(A a, A b) {
      return a.f==b.f; }
   static public bool operator !=(A a, A b) {
      return a.f!=b.f; }
}

A a=new A(); a.f=5; A b=new A(); b.f=5;
A c=new A(); c.f=6;
Console.WriteLine(a==b); // -> True
Console.WriteLine(a==c); // -> False
```

Neue alte Features: Operator Overloading

```
class A {
   int f;
   static public bool operator ==(A a, A b) {
      return a.f==b.f; }
   static public bool operator !=(A a, A b) {
      return a.f!=b.f; }
}
```

```
A a=new A(); a.f=5; A b=new A(); b.f=5;
A c=new A(); c.f=6;
Console.WriteLine(a==b); // -> True
Console.WriteLine(a==c); // -> False
```

Achtung, statische Bindung:

```
object oa=a;
object ob=b;
Console.WriteLine(oa==ob); // -> False
```

Jeweils nur paarweise Überladung von ==, !=, <, >, etc...

Neue alte Features: Parameterübergabe

- Referenz-Parameter

```
void swap(ref int x, ref int y) {
    int temp=x; y=x; x=temp;
}
int a = 0, b = 1;
swap(ref a, ref b); // a = 1, b = 0
```

Neue alte Features: Parameterübergabe

- Referenz-Parameter

```
void swap(ref int x, ref int y) {
    int temp=x; y=x; x=temp;
}
int a = 0, b = 1;
swap(ref a, ref b); // a = 1, b = 0
```

- Rückgabe-Parameter

```
void twice(out int ox, out int oy, int x, int y) {
    ox=x*2;
    oy=y*2;
}
int a, b;
twice(out a, out b,c,d); // a = 2*c, b = 2*d
```

Neue alte Features

- goto

```
if(b) goto foo;
Console.WriteLine("hello");
foo:
Console.WriteLine("world");
```

Neue alte Features

- goto

```
  if(b) goto foo;
  Console.WriteLine("hello");
foo:
  Console.WriteLine("world");
```

  Nur innerhalb von Methoden

Neue alte Features

- einfache Version von Klassen: `struct`

```
struct Point {
  double x, y;
}
```

Structs sind value-Typen

Neue alte Features

- einfache Version von Klassen: `struct`

```
struct Point {
  double x, y;
}
```

Structs sind value-Typen
Speicher auf dem Stack → keine Konstruktoraufrufe

Neue alte Features

- einfache Version von Klassen: `struct`

```
struct Point {
  double x, y;
}
```

Structs sind value-Typen
Speicher auf dem Stack → keine Konstruktoraufrufe
Können bei großen Datenmengen Performance verbessern

Neue alte Features

- einfache Version von Klassen: `struct`

```
struct Point {
  double x, y;
}
```

Structs sind value-Typen
Speicher auf dem Stack → keine Konstruktoraufrufe
Können bei großen Datenmengen Performance verbessern
Teuer, wenn Structs als Parameter übergeben werden!

Neue alte Features

- einfache Version von Klassen: `struct`

```
struct Point {
  double x, y;
}
```

  Structs sind value-Typen
  Speicher auf dem Stack → keine Konstruktoraufrufe
  Können bei großen Datenmengen Performance verbessern
  Teuer, wenn Structs als Parameter übergeben werden!

- echte mehrdimensionale Arrays

```
new int[3][4] // Wie in Java: 4 Arrays (1*3 und 3*4)
new int[3,4] // Ein Array (1*12)
```

Explizites Überschreiben von Methoden

```
class A {
   public virtual void f() {}
}
class C : A {
   public override void f() {}
}
class D : A {
   public new void f() {}
}
```

Explizites Überschreiben von Methoden

```
class A {
   public virtual void f() {}
}
class C : A {
   public override void f() {}
}
class D : A {
   public new void f() {}
}

A a=new D();
a.f();
```

Explizites Überschreiben von Methoden

```
class A {
   public virtual void f() {}
}
class C : A {
   public override void f() {}
}
class D : A {
   public new void f() {}
}

A a=new D();
a.f();  // A.f()
```

Explizites Überschreiben von Methoden

```
class A {
   public virtual void f() {}
}
class C : A {
   public override void f() {}
}
class D : A {
   public new void f() {}
}

A a=new D();
a.f();  // A.f()

a=new C();
a.f(); // C.f()
```

Get-/Set-Methoden

```
class Length {
    double cm;
    public double inches {
        get { return cm * 2.54; }
        set { cm = value/2.54; }
    }
}
```

Get-/Set-Methoden

```
class Length {
    double cm;
    public double inches {
        get { return cm * 2.54; }
        set { cm = value/2.54; }
    }
}

Length a = new Length();
a.inches = 16;        // cm = 6.299...
Console.WriteLine(a.inches); // 16
```

Auch einzeln oder virtuell deklarierbar

Delegates

```
class MyClass {
   Char x;
   public MyClass(Char _x) { x=_x; }
   bool IsEqual(Char c) { return x==c; }
   delegate bool IsAnything (Char c);
   public static void Main () {
      IsAnything validDigit = new IsAnything(Char.IsDigit);

      Console.WriteLine(validDigit('A')); // -> False
      Console.WriteLine(validDigit('4')); // -> True

      MyClass m=new MyClass('5');
      validDigit = new IsAnything(m.IsEqual);
   }
}
```

Parametrischer Polymorphismus

```
class Stack<T> {
    void push(T t) { ... }
    T pop() { ... }
}
```

Parametrischer Polymorphismus

```
class Stack<T> {
   void push(T t) { ... }
   T pop() { ... }
}
...
Stack<int> s;
s.push(67);
int x=s.pop();
s.push("hello"); // -> compiler error
...
```

Parametrischer Polymorphismus

```
class Stack<T> {
   void push(T t) { ... }
   T pop() { ... }
}
...
Stack<int> s;
s.push(67);
int x=s.pop();
s.push("hello"); // -> compiler error
...
```

Prinzipell identisch mit Java 1.5, mächtiger dank Typsystem
Keine Werte als Parameter oder Berechnungsuniversalität (C++)

Attribute

```
public class AuthorAttribute : Attribute {
    private string name;
    private string changed;
    public AuthorAttribute (string name, string changed) {
        this.name = name; this.changed = changed;
    }
    public string author {
        get { return name; }
    }
}
```

Attribute

```
public class AuthorAttribute : Attribute {
    private string name;
    private string changed;
    public AuthorAttribute (string name, string changed) {
        this.name = name; this.changed = changed;
    }
    public string author {
        get { return name; }
    }
}
[Author("Inge Wagner", "1999-10-01")]
public class A {
     [Author("Inge Wagner", "2003-12-03")]
     void f () {}
}
```

Attribute (2)

```
static void Main() {
   foreach(Attribute a in
         Attribute.GetCustomAttributes(typeof(A))) {
      Console.WriteLine(a);
      if(a is AuthorAttribute)
         Console.WriteLine("written by: " +
                           ((AuthorAttribute)a).author);
   }
}
```

Attribute (2)

```
static void Main() {
   foreach(Attribute a in
         Attribute.GetCustomAttributes(typeof(A))) {
      Console.WriteLine(a);
      if(a is AuthorAttribute)
         Console.WriteLine("written by: " +
                           ((AuthorAttribute)a).author);
   }
}
```

Objekte werden erst bei Abfrage wirklich erzeugt ⇒ kein Overhead

Referenzen

- Mono-Projekt
  http://www.mono-project.com
- C#-Spezifikation der ECMA
  http://www.mono-project.com/ECMA

Fragen?