5.– 8. September 2011
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Golos scharfes C

Wissenswertes über C#

Golo Roden
www.goloroden.de

Golos scharfes C

Golo Roden

www.goloroden.de

Über mich

- Wissensvermittler und Technologieberater
  - .NET, Codequalität und agile Methoden
  - MVP für C# in den Jahren 2010 und 2011
  - Agile Development Framework (ADF)
- Autor, Sprecher und Trainer
  - dotnetpro, heise Developer
  - prio.conference, .NET DevCon
- Kontakt
  - www.goloroden.de

.NET für das Web – wirklich?

```
public class Program {
    public static void Main() {
        http://www.goloroden.de
    }
}
```

.NET für das Web – wirklich?

```
public class Program {
    public static void Main() {
        http://www.goloroden.de;
    }
}
```

.NET für das Web – wirklich?

```
public class Program {
    public static void Main() {
        http://www.goloroden.de
        ;
    }
}
```

Nutzen – gleich null?

> Erst das Vergnügen,
dann die Arbeit.

Golos scharfes C

> Welchen Typ hat null?

var – klein und nützlich?

```
var firstName = "Golo";
var age = 32;
var dogs =
    new List<Animal> {
        new Animal { Name = "Alice" },
        new Animal { Name = "Muffin" }
    };
```

Klein und – unschuldig?

```
var foo = null;
```

Unschuldig – welchen Typ hat null?

- ECMA 334, §11.2.7
  - The null literal (§9.4.4.6) evaluates to the **null value**, which is used to denote a reference not pointing at any object or array, or the absence of a value. The **null type** has a single value, which is the null value. Hence an expression whose type is the null type can evaluate only to the null value. There is **no way to explicitly write the null type** and, therefore, no way to use it in a declared type.
  - Moreover, the null type can never be the type inferred for a type parameter (§25.6.4).

Typen – von object abgeleitet?

- null ist eine Anomalie im Typsystem – gibt es weitere?
- Weitere Anomalien
  - TypedReference
    - Laut Reflection **von object** abgeleitet
    - **Nicht nach object konvertierbar**
  - Schnittstellen
    - Laut Reflection **nicht von object** abgeleitet
    - Methoden von **object sind aufrufbar**
- Schlussfolgerung
  - Nicht jeder Typ ist von object abgeleitet, aber fast **jeder Typ kann zu object konvertiert** werden.

Golos scharfes C

> Ein bool ist ein int ist ein bool

Ein bool ist ein int ist ein – bool?

```
if(foo = 23) {
    // ...
}
```

bool – Fakten?

- Typ
  - bool ist ein Alias für **System.Boolean**
  - bool ist in der **mscorlib.dll** enthalten
  - bool ist ein **Wertetyp**
- Wertebereich
  - **true** und **false**
  - bool? als **nullbarer Wertetyp**
- Fakten
  - int ist **nicht implizit** nach bool castbar
- Wie wird ein bool intern dargestellt?
  - 1 Bit? 8 Bit? 32 Bit? 64 Bit? …?

Fakten – unsafe?

- **unsafe**-Code ist „böse“ …
  - … und manchmal notwendig
- Beispiel

```
private unsafe bool ByteToBoolean(byte b) {
    return *((bool*)&b);
}
```

Golos scharfes C

> StringBuilder + String.Concat

StringBuilder + String.Concat = ?

```
var r1 = “Hallo ” + “Welt!”;

var r2 = String.Concat(“Hallo ”, “Welt!”);

var r3 = String.Format(
    “{0} {1}!”, “Hallo”, “Welt”);

var sb = new StringBuilder();
sb.Append(“Hallo ”);
sb.Append(“Welt!”);
var r4 = sb.ToString();
```

string – Fakten?

- Typ
  - string ist ein Alias für **System.String**
  - string ist in der **mscorlib.dll** enthalten
  - string ist ein **Referenztyp**
- Fakten
  - string ist **immutable**
  - string belegt **20 + (n / 2) * 4 Bytes**
- Wie wird ein string intern dargestellt?
  - C
    - **null-terminiert**
  - Pascal
    - Längenangabe als **Prefix**
  - .NET = C + Pascal

Fakten – unveränderbar?

- Immutability
  - Unveränderlichkeit resultiert in **neuen Instanzen**
  - Für identische strings findet in C# **Interning** statt
- Interning
  - **Feature von C#**, nicht von .NET
  - Methode **String.Intern**

Unveränderbar – die Folgen?

- Was folgt aus all dem?
  - +
    - Verketten von **Literalen**, Erhalt der **Zwischenergebnisse**
  - String.Concat
    - Verketten von zahlreichen strings zur **Laufzeit**, innerhalb **eines Aufrufs**
  - String.Format
    - **Langsamste** Variante von allen
  - StringBuilder
    - Verketten von zahlreichen strings zur **Laufzeit**, verteilt über **verschiedene Aufrufe**

Golos scharfes C

> Gleich und doch nicht dasselbe

object – Equals?

```
object
   public static bool ReferenceEquals(
       object first, object second)

   public static bool Equals(
       object first, object second)

   public virtual bool Equals(
       object other)
```

Equals – ReferenceEquals?

- ReferenceEquals
  - Vergleicht, ob die **Referenzen identisch** sind
- Equals (statisch)
  - Vergleicht, ob die **Referenzen identisch** sind – außer bei **Wertetypen**, dort wird der **tatsächliche Wert** verglichen
- Equals (Instanz)
  - Entspricht zunächst der statischen Variante, ist aber **virtual** und kann daher **überschrieben** werden
  - Wird Equals überschrieben, muss zwingend **auch GetHashCode** überschrieben werden
  - Wird Equals überschrieben, sollte bei **Wertetypen auch ==** überschrieben werden

object vs <T>?

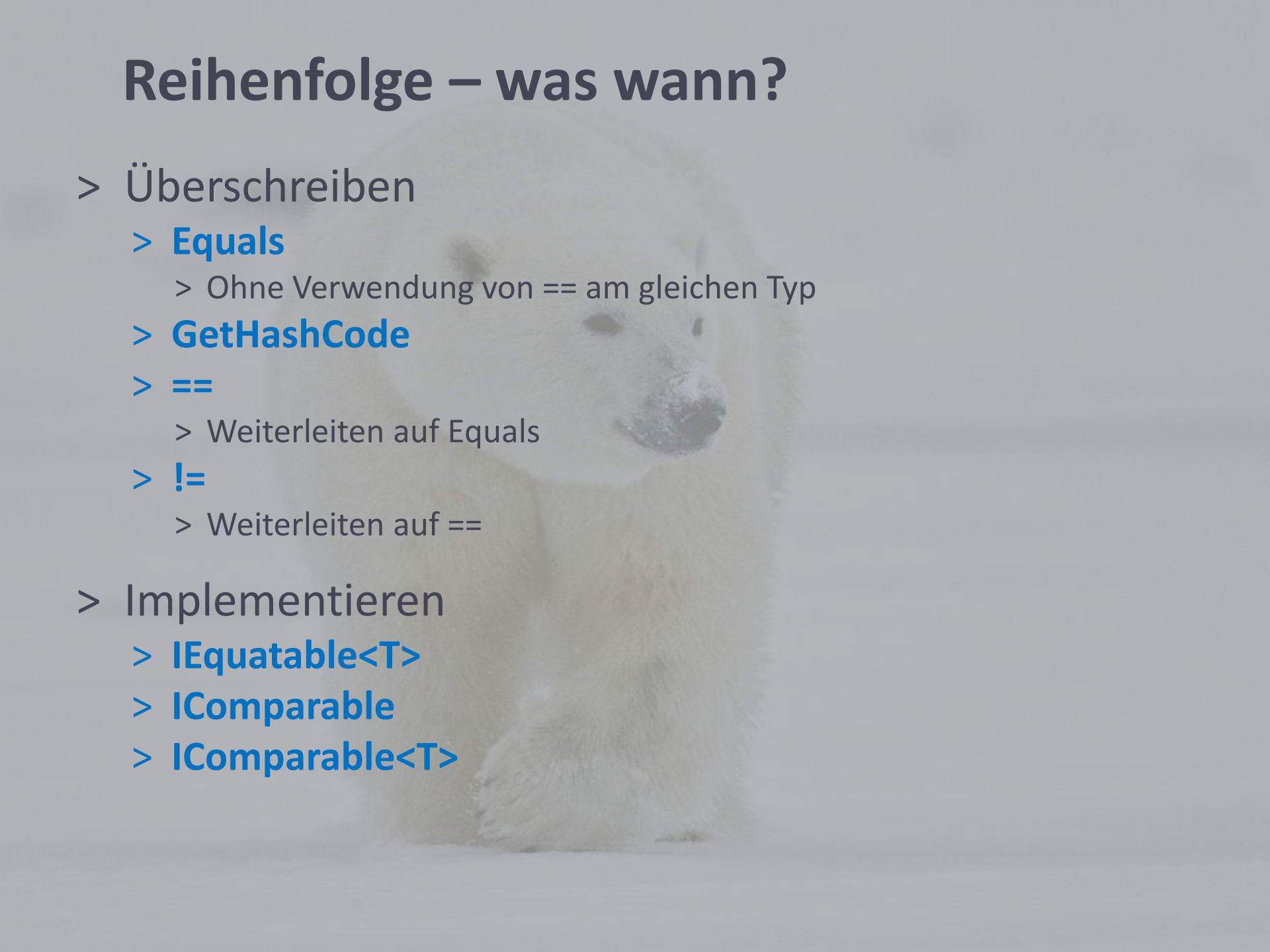
- IEquatable<T>
  - **Equals<T>(T other)**
  - GetHashCode muss nicht überschrieben werden – allerdings ergibt Equals<T> **ohne Equals wenig Sinn**
  - Prinzip der geringsten Überraschung
- IComparable
  - **CompareTo(object other)**
  - Liefert 0, 1 oder -1
- IComparable<T>
  - **CompareTo<T>(T other)**
  - Liefert 0, 1 oder -1

== – überschreiben?

- Wird == überschrieben, so muss **auch !=** überschrieben werden
  - Auf **!(first == second)** weiterleiten
  - **== auf Equals** weiterleiten
- Fallstricke
  - == ist ein **Operator** und Operatoren sind per Definition **statisch**
  - Wird auf eine Instanz **über eine Schnittstelle zugegriffen**, fällt C# stets auf == von object zurück – der überschriebene **Operator wird ignoriert**

Reihenfolge – was wann?

- Überschreiben
  - **Equals**
    - Ohne Verwendung von == am gleichen Typ
  - **GetHashCode**
  - **==**
    - Weiterleiten auf Equals
  - **!=**
    - Weiterleiten auf ==
- Implementieren
  - **IEquatable<T>**
  - **IComparable**
  - **IComparable<T>**



Golos scharfes C

> Richtig Schluss machen

Richtig Schluss machen – bloß wie?

```
// Clean up code
foo = null;
GC.Collect();
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

C++ vs .NET

- Objekte erzeugen
  - C++
    - Freien Speicher suchen
  - .NET
    - **Next Object Pointer**
- Objekt entfernen
  - C++
    - delete-Operator
  - .NET
    - **Garbage Collection**

Finalizer – einfach?

```
public class Foo {
    protected override void Finalize() {
        // Free any unmanaged resources.
        // Always call the base class's
        // finalizer.
        base.Finalize();
    }
}
```

Finalizer – einfacher?

```
public class Foo {
    ~Foo() {
        // Free any unmanaged resources.
    }
}
```

Finalizer – problemlos?

- Probleme
  - **Nicht-deterministische** Ausführung
  - **Performance**
- Lösung
  - Verwendung von **Generationen** in der GC
  - **Drei** Generationen – GC0, GC1 und GC2
- Roots
  - Alle Instanzen, die in **lokalen Variablen referenziert** werden, die also auf dem Stack liegen
  - Alle Instanzen, die **indirekt erreichbar** sind
  - Alle Instanzen, die in der **Finalization Queue** eingetragen sind

GC – die Lösung?

```
// Clean up code
foo = null;
GC.Collect();
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

Die Lösung – IDisposable, Teil 1?

```
public class Foo {
    ~Foo() {
        this.Dispose();
    }

    public void Dispose() {
        // Clean up.
    }
}
```

Die Lösung – IDisposable, Teil 2?

```
public class Foo : IDisposable {
    ~Foo() {
        this.Dispose(false);
    }

    public void Dispose() {
        this.Dispose(true);
    }

    private void Dispose(bool isDisposing) {
        if(isDisposing) {
            // Clean up managed resources.
        }

        // Clean up unmanaged resources.

        GC.SuppressFinalize(this);
    }
}
```

IDisposable – Anmerkungen?

- Weitere Prüfungen
  - Feld **_isDisposed** bei Dispose auf true setzen, und bei jedem weiteren Zugriff eine **ObjectDisposedException** werfen
  - Aufruf von Dispose in **try-finally** kapseln – oder besser, direkt die **using-Anweisung** verwenden
- Finalizer garantieren
  - Die **Ausführung** von Finalizern wird **nicht garantiert**
  - Soll die Ausführung garantiert werden, muss von **CriticalFinalizerObject** abgeleitet werden

Fazit

- Wozu das Ganze?
  - Um auf Developer-Partys Frauen zu beeindrucken?
- Nein – denn es existiert ein Bezug zur Realität
  - **C# ist das primäre alltägliche Handwerkszeug**, kein exotisches Framework
  - **Knifflige Situationen sind Ausnahmen** – aber wenn eine entsprechende Situation eintritt, ist es gut, zu verstehen, wie C# funktioniert und warum

Weiterführende Informationen

- dotnetpro
  - 03.2010 – **Welchen Typ hat null?**
  - 04.2010 – **Ein bool ist ein int ist ein bool**
  - 05.2010 – yield return, yield break, yield …
  - 06.2010 – beforefieldinit
  - 07.2010 – V … wie virtual
  - 08.2010 – **Gleich und doch nicht dasselbe**
  - 09.2010 – Operator Overloading 101
  - 10.2010 – IArithmetic
  - 11.2010 – (K)eine teure Angelegenheit
  - 12.2010 – **StringBuilder + String.Concat**
  - 01.2011 – **Richtig Schluss machen, Teil 1**
  - 02.2011 – **Richtig Schluss machen, Teil 2**

Wird fortgesetzt ab dotnetpro 09.2011!

Feedback

> Fragen, Anregungen, Lob oder Kritik?

> www.goloroden.de