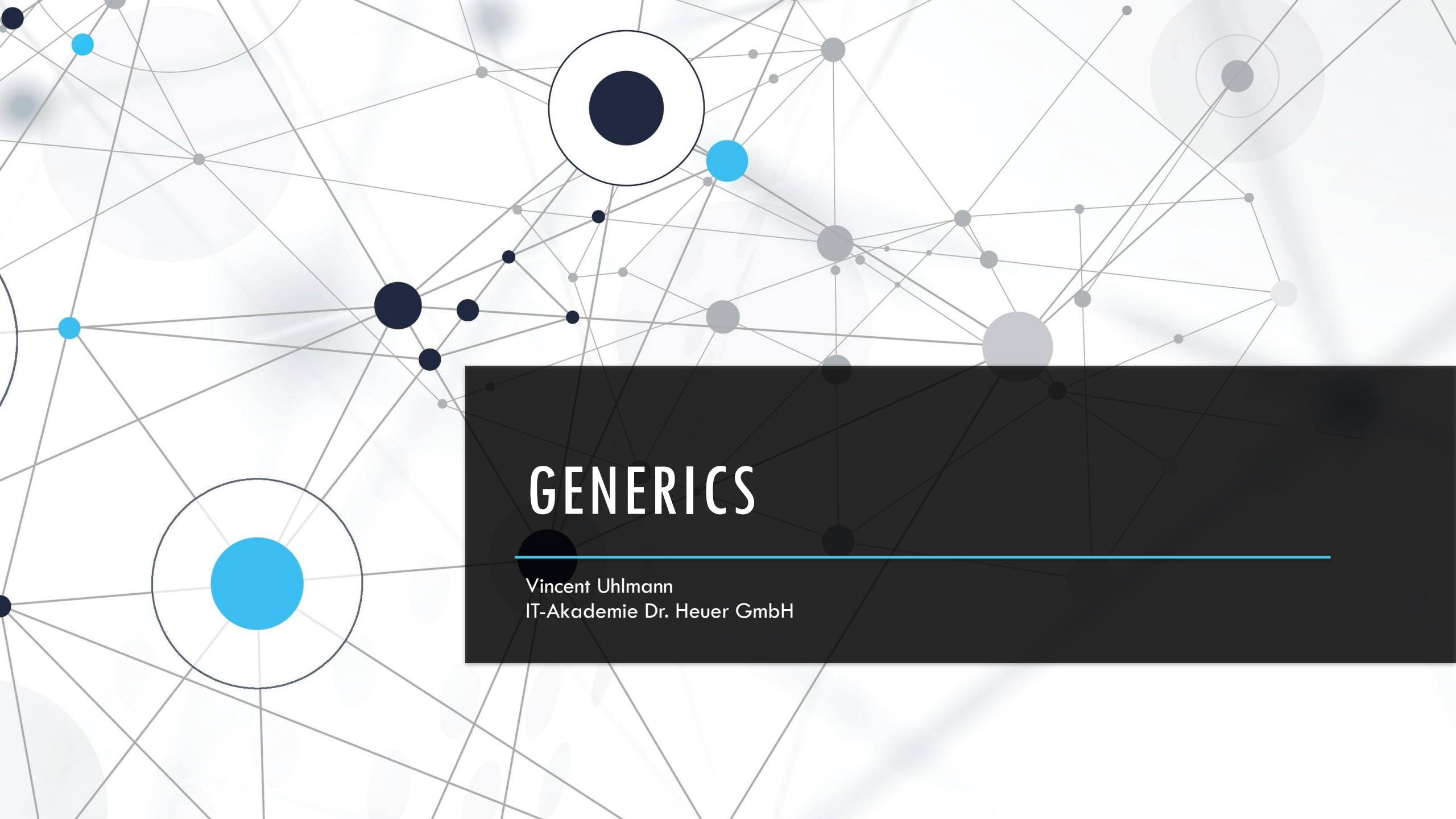

# GENERICS

Vincent Uhlmann
IT-Akademie Dr. Heuer GmbH

# PROBLEM

- Wenn wir eine Klasse wie z. B. eine Liste erstellen möchten, die mit mehreren Datentypen arbeiten kann, haben wir mehrere Möglichkeiten

- Wir könnten für jeden Datentyp eine eigene Klasse erstellen (IntList, DoubleList, BoolList…)
- Sorgt für sehr viele Klassen und zu viel gleichen Code

- Wir könnten eine Liste von Objekten erstellen, da jeder Datentyp als Objekt betrachtet werden kann (ObjektList)
- Wäre sehr umständlich und nicht typsicher (siehe z.B. ArrayList)

# GENERICS

- Platzhalter für den Datentyp
- Wird zur Laufzeit durch einen konkreten Typ ersetzt
- Erhöht Typsicherheit und Wiederverwendbarkeit des Codes
- Erhöhen die Leistung da kein Boxing und Unboxing erforderlich ist
- Der Name ist frei wählbar, beginnt aber mit einem großen T (T, TValue...)
- Kann z.B. für Klassen, Interfaces, Methoden, Eigenschaften, Ereignisse oder Delegaten angegeben werden
- Es ist möglich, mehr als einen Platzhalter anzugeben

# SYNTAX

- Typparameter in spitzen Klammern nach dem Typnamen

```
public static void Test<TValue> (TValue value)
{
    Console.WriteLine($"Wert: {value}");
}
```

```
public class DataStore<T>
{
    private readonly T[] _arr;

    public DataStore(T[] arr)
    {
        _arr = arr;
    }

    public void PrintAllItems()
    {
        foreach(var item in _arr)
        {
            Console.WriteLine(item);
        }
    }
}
```

# CONSTRAINTS (EINSCHRÄNKUNGEN)

- Durch Constraints können Generische Typen eingeschränkt werden
- Zum Beispiel kann der Zugriff auf eine Methode nur für Typen erlaubt sein, die von einer bestimmten Klasse erben oder eine bestimmte Schnittstelle implementieren
- Keyword where

```
public static void Test<TValue> (TValue value) where TValue : IEnumerable<int>
{
    Console.WriteLine($"Wert: {value.Count()}");
}
```

# CONSTRAINTS (EINSCHRÄNKUNGEN)

```
// T muss ein Referenztyp (Klasse) sein
public static void Test1<T>(T value) where T : class
{
}

// T muss ein Wertetyp (int, double oder eigenes Struct) sein
public static void Test2<T>(T value) where T : struct
{
}

// T muss das Interface IList implementieren
public static void Test3<T>(T value) where T : IList
{
}

// T muss U sein oder davon Ableiten (Vererbung)
public static void Test4<T, U>(T value) where T : U
{
}
```

# DEFAULT

- Das Schlüsselwort default erzeugt den Standardwert des konkreten Typs
- Kann für Variablen, Rückgabeanweisungen oder Standardwerte eines Parameters verwendet werden
- Liefert bei Referenztypen null zurück
- Bei Wertetypen wie int, double … den Wert 0, sonst der Standard des Datentyps

```
public static TValue? Test<TValue> (TValue value, TValue? value2 = default)
{
    return default;
}
```

# UMWANDLUNG

- T kann in den expliziten Typ konvertiert werden
- Nützlich wenn z.B. Methoden des expliziten Typs aufgerufen werden sollen
- Laufzeitfehler sollte T nicht in den spezifischen Typ umgewandelt werden können

```
public static void Foo<T>(T value) where T : Person
{
    if (value is Mitarbeiter)
    {
        Mitarbeiter mitarbeiter = (Mitarbeiter)(object)value;
        mitarbeiter.MitarbeiterFunktion();
    }
    else if (value is Praktikant)
    {
        Praktikant praktikant = (Praktikant)(object)value;
        praktikant.PraktikantFunktion();
    }
}
```

# VERERBUNG

- Wenn generische Basisklassen vererbt werden, wird der Typ durch die Oberklasse festgelegt
- Es besteht auch die Möglichkeit, dass eine nicht-generische Oberklasse von einer generischen Basisklasse erbt und umgekehrt

```
public class Mitarbeiter<T>
{
    public Mitarbeiter(T value)
    {
    }
}

public class Praktikant<T> : Mitarbeiter<T>
{
    public Praktikant(T value) : base(value)
    {
    }
}
```

# HINWEISE

- Der Typ kann beim Aufruf einer generischen Methode weggelassen werden, wenn er sich aus den Parametern ergibt

```
Foo(5);

public static void Foo<T>(T value)
{
}
```

- Das Schlüsselwort default kann verwendet werden, um eine generische Methode mit dem Standardwert des Typs aufzurufen
- In diesem Fall muss der Typ angegeben werden

```
Foo<int>(default);

public static void Foo<T>(T value)
{
}
```