Vorlesung Softwareentwicklung 2021

https://github.com/SebastianZug/CsharpCourse

André Dietrich Christoph Pooch Fabian Bär Fritz Apelt Galina Rudolf JohannaKlinke Jonas Treumer KoKoKotlin Lesestein LinaTeumer MMachel Sebastian Zug Snikker123 Yannik Höll Florian2501 DEVensiv fb89zila

Generics

Parameter	Kursinformationen
Veranstaltung:	`Vorlesung Softwareentwicklung`
Semester	`Sommersemester 2021`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Generics und deren Anwendung`
Link auf den GitHub:	https://github.com/TUBAF-IfI-LiaScript/VL_Softwareentwicklung/blob/master/19_Generics.md
Autoren	@author

Motivation

Versuchen wir nach dem Exkurs zur UML Modellierung und den Werkzeugen, den roten Faden der C# Programmierung wieder aufzunehmen. In dieser Woche wollen wir uns mit der Frage der *Generics* und damit der Frage beschäftigen, wie wir ohne eine spezifische Berücksichtigung von Datentypen wiederverwendbareren Code schreiben können.

Nehmen wir an, dass Sie ohne die entsprechenden .NET-Bibliotheken eine Liste für `int`-Werte implementieren sollen. Warum funktioniert das Ganze nicht mit unserem bisherigen Array-Konzept?

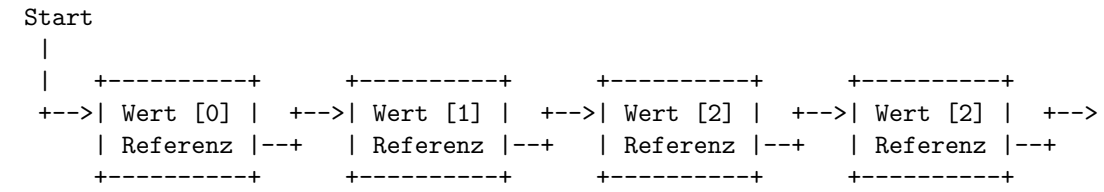
```
using System;
using System.Reflection;
using System.Reflection.Emit;

public class Program{
    public static void Main(string[] args){
      var myArray = new[] { 1, 2, 3, 4, 5 };
      foreach (int i in myArray)
      {
          System.Console.Write("{0} ", i);
      }
      Type valueType = myArray.GetType();
      Console.WriteLine("\nmyArray: Type is {0}", valueType);
      Console.WriteLine("\nmyArray is Array? {0}", valueType.IsArray);
      //myArray.Add(7);
    }
}
```

Die Dokumentation von `Array` findet sich unter https://docs.microsoft.com/de-de/dotnet/api/system.array?view=netcore-3.1

Merke: Arrays sind in C# statisch definiert und haben keine veränderliche Größe.

Lassen Sie uns einen alternativen Ansatz bestreiten. Wir implementieren ein eigenes Konzept, dass eine verkettete Liste repräsentiert.




```
using System;

public class Node{
    public Node next;
    public int value;
    public Node(Node next, int value){
        this.next = next;
        this.value = value;
    }
}

public class LinkedList{
    public Node head;
    public LinkedList(int initial) {
        head = new Node(null, initial);
    }

  public void Add(int value){
      Node current = head;
      while (current.next != null){
          current = current.next;
      }
      current.next = new Node(null, value);
  }

  public int this[int index]{
      get {
          Node current = head;
          int count = 0;
          while (current != null){
              if (count == index){
                 return current.value;
              }
              current = current.next;
              count++;
          }
          return 1;
      }
   }
}

public class Program{
    public static void Main(string[] args){
      LinkedList linkedList = new LinkedList(121);
      linkedList.Add(140);
      linkedList.Add(280);
      linkedList.Add(309);
      int i = 2;
      Console.WriteLine($"Der Wert des {i}. Eintages ist {linkedList[i]}.");
    }
```

}

Was sind die Nachteile in dieses Konstrukts auf der Listenebene? Welche Lösungsansätze sehen Sie?

Im Hinblick auf die Wiederverwendbarkeit stellt sich noch eine weiteres Problem - die Lösung ist typabhängig, die Speicherung eines anderen Datentypen macht eine Neuimplementierung notwendig. Zählen Sie doch mal durch, wie oft wir aus dem `int` ein `float` machen müssten, um eine Übertragbarkeit auf Fließkommazahlen zu realisieren. Damit entstünde dann aber auch ein überwiegend redundanter Code, der eine konsistente Realisierung und Wartung erheblich erschwert.

Lösungsansatz könnte die Arbeit mit dem allgemeinen `Object`-Datentyp sein. Mittels Boxing und Unboxing würden die spezifischen Datentypen auf diesen abgebildet.

> Merke (Wiederholung): Alle C# Datentypen sind von `Object` abgeleitet.

```
int i = 123;
object o = i;  // The following line boxes i.

o = 123;
i = (int)o;    // unboxing
```

Nachteilig daran ist, dass

- diese Operation Laufzeit kostet,
- beim Auslesen der Daten eine externe (außerhalb unserer Liste liegende) Cast-Operation erforderlich macht. `float x = (float) linkedList[i]`,
- die Klasse würde alle Datentypen akzeptieren. Unter Umständen ist das aber nicht gewünscht weil zum Beispiel mit Zahlenwerten arithmetische Operationen ausgeführt werden sollen. Eine Beschränkung ist aber nicht möglich.

```
using System;

public class Cat{
  public void catchMouse(){
    Console.WriteLine("Dies kann allein die Katze!");
  }
  public void makeSound(){
    Console.WriteLine("Miau");
  }
}

public class Dog{
  public void huntCat(){
    Console.WriteLine("Dies kann allein der Hund!");
  }
  public void makeSound(){
    Console.WriteLine("Wuff");
  }
}

public class Program{
  public static void Main(string[] args){
    Cat Kitty = new Cat();
    Dog Wally = new Dog();
    Object Animal = Kitty;
    (Animal as Cat).catchMouse();
  }
}
```

> *"Of course, we love bugs … but not on run-time!"* (Youtube Tutorial Generics in .NET)

Generische Typen

"Generics" sind seit der Version 2.0 Elemente der .Net-Sprachen und der Common Language Runtime (CLR). Sie definieren das Konzept der Typparameter, wodurch Klassen und Methoden keiner konkreten Zuordnung zu

einem Datentyp unterworfen werden. Platzhalter übernehmen die generische Repräsentation des Typen, die Ersetzung erfolgt zur Laufzeit.

```
// Generische Klassenspezifikation
public class LinkedList<T>{
  public void Add(T value){
    ...
  }
  public T this[int index]{
    ...
  }
}

// Instanzierung mit verschiedenen Datentypen
LinkedList<float> list1 = new LinkedList<float>(3.14);
LinkedList<ExampleClass> list2 = new LinkedList<ExampleClass>(myExampleClass);
LinkedList<ExampleStruct> list3 = new LinkedList<ExampleStruct>(myExampleStruct);
```

Die Vorteile des Konzepts sind offensichtlich:

- Der Compiler kann eine spezifische Typprüfung durchführen.
- Die Operationen sind effektiver, weil keine Typumwandlungen (wie beim Umweg über `Object`) realisiert werden müssen.
- Programme werden lesbarer.

Generische Klassen und Methoden vereinen Wiederverwendbarkeit, Typsicherheit und Effizienz so, wie es ihre nicht generischen Gegenstücke nicht können. Generics werden am häufigsten für Auflistungen und deren Methoden verwendet.

Was passiert eigentlich hinter den Kulissen? Im Unterschied zu C++ Templates werden C# Generics nicht zur Compile-Zeit konkretisiert, sondern zunächst in einen generischen Zwischencode übersetzt. Die eigentliche Konkretisierung findet zur Laufzeit statt, wobei Referenz- und Wertdatentypen unterschiedlich behandelt werden. Für jeden Werttyp, der den Platzhalter ersetzt wird eine konkrete Klasse erzeugt, während sich alle Referenztypen eine einzige Konkretisierung teilen. Das Laufzeitsystem erzeugt den typentsprechenden Code erst erst mit ersten Instanzierung der konkreten Klasse.

Die Parameterisierung eines generischen Typs beschränkt sich nicht nur auf einen Typ sondern kann mehrere Typen umfassen.

```
class MyGenericClass<T, U>
{
    ...
}
```

Hinsichtlich der Namenswahl für die generischen Typen sind sie frei, sollten aber berücksichtigen, dass für den Leser ggf. unklar ist, wie welcher konkrete Datentyp realisiert werden kann. Die Einbuchstabenvariante “T” sollte nur genutzt werden, wenn in Bezug auf einen Container die Bedeutung wirklich klar ist.

```
using System;

public class Stack<T>{
  int position = 0;
  T[] data = new T[100];

  public void Push(T newObj){
    if (position < 100){
      data[position++] = newObj;
    }
    else{
      Console.WriteLine("Array size exceeded!");
    }
  }

  public T Pop(){
    return data[position--];
```

```
  }

  public override string ToString(){
    string output = "";
    for (int i=0; i<position; i++){
       output = output + " " + data[i].ToString();
    }
    return output;
  }
}

public class Program{
    public static void Main(string[] args){
      var myStack = new Stack<int>();
      myStack.Push(3);
      myStack.Push(12);
      //myStack.Push("Hallo!");
      Console.WriteLine(myStack);
   }
}
```

Generische Methoden

```
using System;

public class Program{

  // Tauscht zwei Variablen lhs und rhs
  static void Swap<T>(ref T lhs, ref T rhs)
  {
     T temp;
     temp = lhs;
     lhs = rhs;
     rhs = temp;
     Console.WriteLine("Hier wurde die generische Methode aufgerufen");
  }

  // Tauscht zwei Variablen lhs und rhs
  static void Swap(ref int lhs, ref int rhs)
  {
     int temp;
     temp = lhs;
     lhs = rhs;
     rhs = temp;
     Console.WriteLine("Hier wurde die konkrete Methode aufgerufen");
  }

  public static void Main(string[] args){
     int a = 99;
     int b = 1;
     //    ^
     //    ------ Abstimmung der Typen
     //    v
     Swap<int>(ref a, ref b);
     System.Console.WriteLine("a=" + a + " ,b=" + b);
     Swap(ref a, ref b);
     System.Console.WriteLine("a=" + a + " ,b=" + b);
     float x = 99F;
     float y = 1.2345F;
     Swap<float>(ref x, ref y);
     System.Console.WriteLine("x=" + x + " ,y=" + y);
```

```
  }
}
```

Sie können das Typargument auch weglassen, der Compiler löst den Typ entsprechend auf. Eine Einschränkung oder ein Rückgabewert genügen ihm zur Ableitung des Typparameters nicht. Damit ist ein Typrückschluss bei Methoden ohne Parameter nicht möglich! Damit bewirken:

```
Swap<int>(ref a, ref b);  // und
Swap(ref a, ref b);
```

einen analogen Aufruf.

Welches Fragestellungen ergeben sich aus dem Codefragment:

- Was passiert, wenn eine "identische" nicht-generische Methode bereitsteht? (vgl. Zeile 18 in obigem Beispiel)
- Welche Probleme entstehen, wenn die generische Methode in eine generische Klasse eingefügt wird?
- Wie stellen wir sicher, dass spezifische Methoden für den Datentyp überhaupt existieren?

Die erste Frage lässt sich schnell beantworten, in diesem Fall wird die nicht-generische Methode aus Effizienzgründen vorgezogen.

```
public class DoAnything<T>{

  // Tauscht zwei Variablen lhs und rhs
  static void Swap<T>(ref T lhs, ref T rhs)
  {
       T temp;
       temp = lhs;
       lhs = rhs;
       rhs = temp;
  }

  ...
}
```

Wenn eine generische Methode definiert wird, die die gleichen Typparameter wie die übergeordnete Klasse verwendet (hier `T`), gibt der Compiler die Warnung CS0693 aus. Innerhalb des Gültigkeitsbereichs der Methode wird der "äußere Klassentyp" durch den "inneren Methodentyp" ausgeblendet. Damit soll der Entwickler, der ggf. zwei unterschiedliche Typen avisiert darauf hingewiesen werden, dass diese hier keine Berücksichtigung finden.

"' .NET Dokumentation Compilerwarnung (Stufe 3) CS0693

Der Typparameter "Typparameter" hat denselben Namen wie der Typparameter des äußeren Typs "Typ".

Dieser Fehler tritt bei einem generischen Member, z. B. einer Methode in einer generischen Klasse, auf. Da der Typparameter der Methode nicht notwendigerweise mit dem Typparameter der Klasse übereinstimmt, können Sie ihm nicht den gleichen Namen geben. Weitere Informationen finden Sie unter Generic Methods (Generische Methoden).

Um diese Situation zu vermeiden, verwenden Sie für einen der Typparameter einen anderen Namen. "'

Verwenden Sie Beschränkungen, analog zu den generischen Typen, sinnvolle Einschränkungen für die Typparametern in Methoden gewährleisten. Das folgende Beispiel gibt als Beschränkung die Implementierung des Interfaces IComparable an, um unseren Vergleich zu realisieren.

```
using System;

public class Program{

  static void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T> {
    T temp;
    if (lhs.CompareTo(rhs) > 0)
    {
        temp = lhs;
        lhs = rhs;
        rhs = temp;
```

```
    }
}

public static void Main(string[] args){
    int a = 99;
    int b = 1;
    SwapIfGreater<int>(ref a, ref b);
    System.Console.WriteLine("a=" + a + " ,b=" + b);
  }
}
```

Was verbirgt sich hinter dem Interface `IComparable`? Werfen Sie einen Blick auf die entsprechende Dokumentation und benennen Sie die Methoden, die in Klassen, die dieses Interface implementieren, existieren müssen.

https://docs.microsoft.com/de-de/dotnet/api/system.icomparable?view=netframework-4.8

> Achtung: Dieses Beispiel benutzt die typbehaftete Variante des IComparable Interfaces! Diese generiert über das Boxing und Unboxing einen unnötigen Aufwand und ist wie zu Beginn gezeigt nicht typsicher. Ersetzen Sie IComparable durch Icomparable .

```
using System;

public class Animal : IComparable {
  private string name;
  private int weight;

  public Animal(string name, int weight){
    this.name = name;
    this.weight = weight;
  }

  public string Name{
    get { return name;}
  }

  public int Weight{
    get { return weight;}
  }

  public override string ToString(){
    return name + " weights " + weight + " kg";
  }

  public int CompareTo (object obj){
    if (obj == null)
       throw new ArgumentException("Object is not a valid");
    else {
      Animal otherAnimal = obj as Animal;
      return (otherAnimal.weight - weight);
      }
    }
  }

public class Program{

  static void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable{
    T temp;
    if (lhs.CompareTo(rhs) > 0)
    {
        temp = lhs;
        lhs = rhs;
        rhs = temp;
    }
```

```
  }

  public static void Main(string[] args){
    Animal AnimalA = new Animal("Kitty", 10);
    Console.WriteLine(AnimalA);
    Animal AnimalB = new Animal("Wally", 30);
    Console.WriteLine(AnimalB);
    SwapIfGreater<Animal>(ref AnimalA, ref AnimalB);
    Console.WriteLine("After ordering ...");
    Console.WriteLine(AnimalA);
    Console.WriteLine(AnimalB);
  }
}
```

Beschränkungen

Wie bereits bei den generischen Methoden angedeutet können wir mittels "Beschränkungen" sicherstellen, dass eine gültige Operation für einen Datentyp existiert.

```
using System;

public class Program{

  static int Plus<T>(T x, T y){
     return (x + y);
  }

  public static void Main(string[] args){
    int a = 1;
    int b = 2;
    Console.WriteLine(Plus<int>(a, b));
  }
}
```

Folglich ist es notwendig die Allgemeinheit der generischen Methoden oder Klassen zu beschränken. Man definiert Beschänkungen oder *Constraints*, die die Breite der verwendbaren Datentypen einschränken. Die Typprüfung bezieht diese Informationen dann ein.

Beschränkung	Das Typargument muss …
`where T : struct`	… ein Werttyp sein.
`where T : class`	… ein Verweistyp sein.
`where T : <Basisklasse>`	… die Basisklasse sein oder von ihr abgeleitete sein.
`where T : <Schnittstelle>`	… die Schnittstelle sein oder diese implementieren.

Das folgende Beispiel setzt die Möglichkeiten der Beschränkung konsequent um und lässt nur `Employee` selbst oder abgeleitete Typen zu. Damit wird sichergestellt, dass die Methoden, die in GenericList verwendet werden, im Parametertypen auch existieren.

```
public class Human
{
    public Employee(string s, int i) => (Name, ID) = (s, i);
    public string Name { get; set; }
    public int ID { get; set; }
}

public class Employee : Human
{
...
}

public class Customer : Human
```

```
{
...
}

public class GenericList<T> where T : Human, IComparable {

    // hier werden Methoden oder Felder der Klasse Employee unter Ausnutzung
    // der Vergleichbarkeit genutzt.

}
```

Da die Konkretisierung von Generics erst zur Laufzeit realisiert werden, ist es ggf. notwendig die spezifischen Parameter des Datentyps zur Laufzeit auszuwerten. Im folgenden sollen die Beispiele die Bedeutung diese Vorgehens aufzeigen.

```
using System;
using System.Reflection;

public class Base {}

class SampleClass<T> where T : Base
{
  void Swap(ref T lhs, ref T rhs) { }
}

public class Program{

  public static void Main(string[] args){
    Type t = typeof(SampleClass<>);
    Console.WriteLine("Liegt ein generischer Typ vor? {0}",
                t.IsGenericTypeDefinition);
    Console.WriteLine("Wie ist der Typparameter benannt? {0}",
                            t.GetGenericArguments());

    Type[] defparams = t.GetGenericArguments();
    foreach (Type tp in defparams){
      Console.WriteLine("\r\nType parameter: {0}", tp.Name);
      Type[] tpConstraints = tp.GetGenericParameterConstraints();
      foreach (Type constr in tpConstraints){
        Console.WriteLine("\t{0}", constr);
      }
    }
  }
}
```

Vererbung bei generischen Typen

In der UML werden generische Typen über eine separate Box in der oberen linken Ecke der Klassendarstellung im Klassendiagramm realisiert.

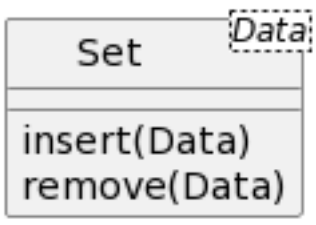


Figure 1: UseCaseOnlineShopII

Generischen Typen können wie andere Typen von einer Klasse erben und Interfaces implementieren. Die

Basisklassen können dabei selbst wieder generische sein.

Ableitung	Die generische Klasse A erbt …	Bemerkung
`class A<X>: B {}`	… vom konkreten Typ B.	
`class A<X>: B<int> {}`	… vom konkretisierten generischen Typ B	
`class A<X>: B<X> {}`	… vom generischen Typ mit gleichem Platzhalter	
`class A: B<X> {...}`		nicht erlaubt!

Beispiele

```
class BaseNode { }
class BaseNodeGeneric<T> { }

class NodeConcrete<T> : BaseNode { }             // concrete type
class NodeClosed<T> : BaseNodeGeneric<int> { }  //closed constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }       //open constructed type
```

Spannend wird die Typparameterisierung für generische Klassen, die von offenen konstruierten Typen erben. Hier müssen für sämtliche Basisklassen-Typparameter Typargumente bereitgestellt werden.

```
class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> { }
```

Im letzten Fall kann ausgehend von der Spezifikation von `Node6<int> A = new Node6<int>();` der Compiler nicht auf die konkrete Realisierung von `U` schließen.

Anwendung

Erläutern Sie die Anwendung von generischen Typen anhand des folgenden UML-Diagramms.

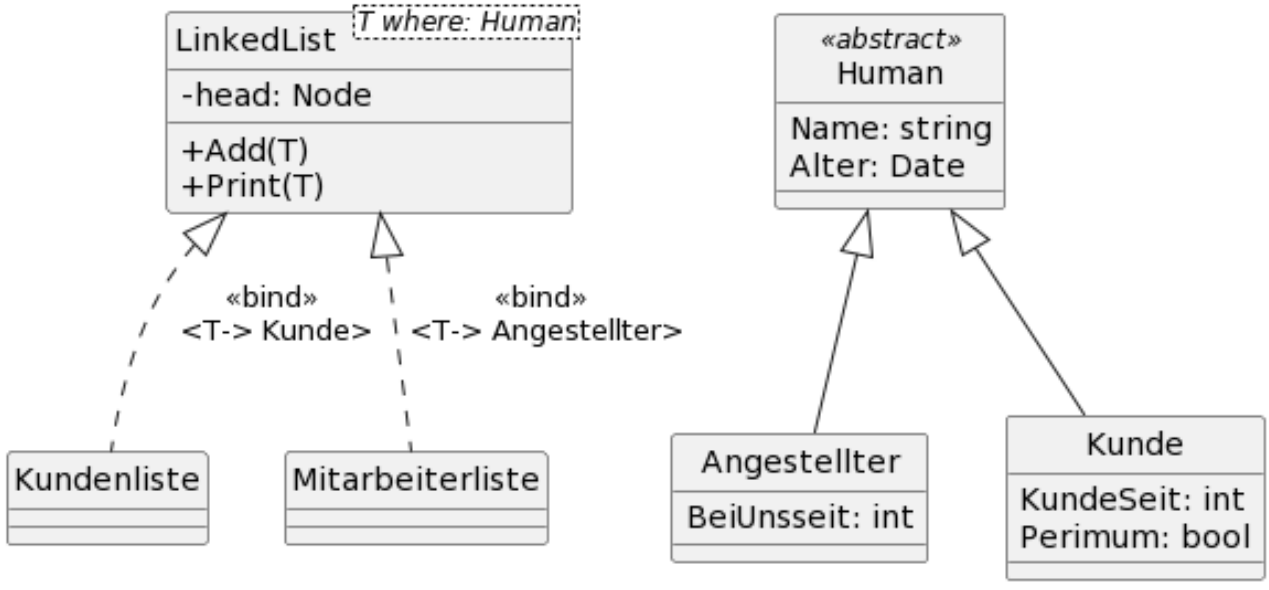


Figure 2: ObjectBasedList

Im folgenden Beispiel soll eine Liste für zwei Klassen dienen, die unterschiedliche Felder umfassen. Die Werte der Felder und die Feldnamen sollen ausgegeben werden.

Anmerkung: In der verwendeten Konfiguration gibt `GetType().GetFields()` lediglich die als public markierten Felder zurück. Entsprechend wurden diese in Angestellter und Kunde definiert.

```
using System;
using System.Reflection;

public abstract class Human{
   public Human (string name, int alter){
      this.Name = name;
      this.Alter = alter;
   }
   public string Name;
   public int Alter;
}

public class Angestellter : Human{
   public int BeiUnsSeit = 0;
   public Angestellter(string name, int alter, int beiUnsSeit) : base(name, alter){
     this.BeiUnsSeit = beiUnsSeit;
   }
}

public class Kunde : Human{
   public int KundeSeit = 0;
   public bool Premium = true;
   public Kunde(string name, int alter, int kundeSeit) : base(name, alter){
     this.KundeSeit = kundeSeit;
   }
}

class LinkedList<T> where T : Human
{
  private class Node{
      public Node next;
      public T data;
      public Node(Node next, T data){
          this.next = next;
          this.data = data;
      }
  }

  private Node head;
  public LinkedList(T initial) {
      head = new Node(null, initial);
  }

  public void Add(T value){
      Node current = head;
      while (current.next != null){
          current = current.next;
      }
      current.next = new Node(null, value);
  }

  public void Print(){
      Node current = head;
      var Fields = current.data.GetType().GetFields();
      foreach (var Field in Fields)
        Console.Write("{0,-20}",Field.Name);
      Console.WriteLine();
      while (true) {
```

```
            Fields = current.data.GetType().GetFields();
            foreach (var Field in Fields){
              Console.Write("{0,-20}", Field.GetValue(current.data));
            }
            Console.WriteLine();
            if (current.next == null) break;
            current = current.next;
        }
    }
}

public class Program{
    public static void Main(string[] args){
      LinkedList<Angestellter> AngestelltenListe = new LinkedList<Angestellter>(new Angestellter("Peter",
      AngestelltenListe.Add(new Angestellter("Viola", 39, 14));
      AngestelltenListe.Add(new Angestellter("Miriam", 32, 5));
      AngestelltenListe.Print();
      Console.WriteLine();
      LinkedList<Kunde> KundeListe = new LinkedList<Kunde>(new Kunde("Garfield", 12, 4));
      KundeListe.Add(new Kunde("Bart", 5, 14));
      KundeListe.Add(new Kunde("Tim", 19, 5));
      KundeListe.Print();
    }
}
```

Aufgaben der Woche

- [ ] Integrieren Sie in die oben gezeigte Liste Suchfunktionen, die nach bestimmten Namen oder Typen filtert.