COLLECTIONS

Vincent Uhlmann
IT-Akademie Dr. Heuer GmbH

COLLECTIONS

- In C# können wir Arrays als auch Collections verwenden, um mehrere Werte zu speichern

- Arrays werden mit einer festen Größe erstellt und können im Nachhinein nicht mehr geändert werden
- Die Elemente in einem Array müssen vom selben Datentyp sein

- Collections wie Listen oder Dictionaries sind im Vergleich dazu dynamisch
- Sie vergrößern oder verkleinern ihre Kapazität entsprechend der Einträge

GENERISCHE VS NICHT-GENERISCHE COLLECTIONS

- Es gibt zwei Arten von Collections in C#, die generischen und die nicht-generischen Collections

- Generische Collections liegen im Namespace System.Collections.Generic
- Sie sind aufgrund der Typsicherheit zu bevorzugen

- Nicht generische Collections liegen im Namespace System.Collections

GENERISCHE VS NICHT-GENERISCHE COLLECTIONS

- Collections

- ArrayList
- Stack
- Queue
- Hashtable

- Generische Collections

- List<T>
- Stack<T>
- Queue<T>
- Dictionary<TKey, TValue>

YIELD

- yield ermöglicht eine Iteration über eine Collection wie ein Array oder eine Liste
- Vermeidet dabei die Notwendigkeit einer temporären Collection
- Ermöglicht saubere und lesbare Implementierung von Iteratoren
- Die Kontrolle kehrt kurzzeitig an den Aufrufer zurück
- Die Iterator Methode führt bei Rückkehr die Anweisungen nach dem yield Schlüsselwort aus

```
static void Main(string[] args)
{
    List<int> list = new List<int>();
    list.Add(1);
    list.Add(2);
    list.Add(3);
    list.Add(4);

    foreach(var x in Sort(list))
        Console.WriteLine(x);
}

public static IEnumerable<int> Sort(List<int> list)
{
    foreach(int i in list)
    {
        if (i % 2 == 0)
            yield return i;
    }
}
```

IENUMERATOR

- IEnumerator ist eine Schnitstelle, die Methoden und Eigenschaften bereit stellt, um eine Collection zu durchlaufen
- Wird verwendet, um Logik für die Enumeration einer Collection zu definieren
- Wird von Klassen wie List<T>, Queue<T> oder Stack<T> implementiert

- Methoden
  - MoveNext(): Bewegt den Enumerator zum nächsten Element in der Sammlung
  - Reset(): Setzt den Enumerator auf seine Anfangsposition zurück

- Eigenschaften
  - Current: Gibt das aktuelle Element in der Sammlung zurück

IENUMERABLE

- IEnumerable ist eine Schnittstelle, die eine Methode GetEnumerator() definiert, die einen IEnumerator zurück gibt
- Sie wird verwendet, um zu bestimmen, ob eine Sammlung enumerierbar ist

- Methode:
  - GetEnumerator()

ARRAYLIST

- Nicht generische Collection
- Dynamisches Array von Werten
- Zugriff über den Index, z.B. arrayList[index]

```
ArrayList arrayList = new ArrayList();
arrayList.Add(5);
arrayList.Add("Hallo");
arrayList.Add(new Program());
```

STACK

- Nicht generische Collection
- LIFO (Last-In-First-Out) Datenstruktur
- Zuletzt eingefügtes Element wird als erstes entfernt
- Dynamische Größte

```
Stack stack = new Stack();
stack.Push(5);
stack.Push("Hallo");

object? o = stack.Pop();

if(o != null && o is string)
    Console.WriteLine((string) o);
```

QUEUE

- Nicht generische Collection
- FIFO (First-In-First-Out) Datenstruktur
- Zuerst eingefügtes Element wird als erstes entfernt
- Dynamische Größte

```
Queue queue = new Queue();
queue.Enqueue(5);
queue.Enqueue("Peter");
object? o = queue.Dequeue();

if(o != null && o is int)
{
    int i = (int)o;
    Console.WriteLine(i * i);
}
```

HASHTABLE

- Nicht generische Collection
- Speichert Schlüssel-Wert-Paare
- Optimiert Suchvorgänge durch Hashing der Schlüssel
- Schlüssel: Eindeutig und nicht null
- Werte: Können null sein

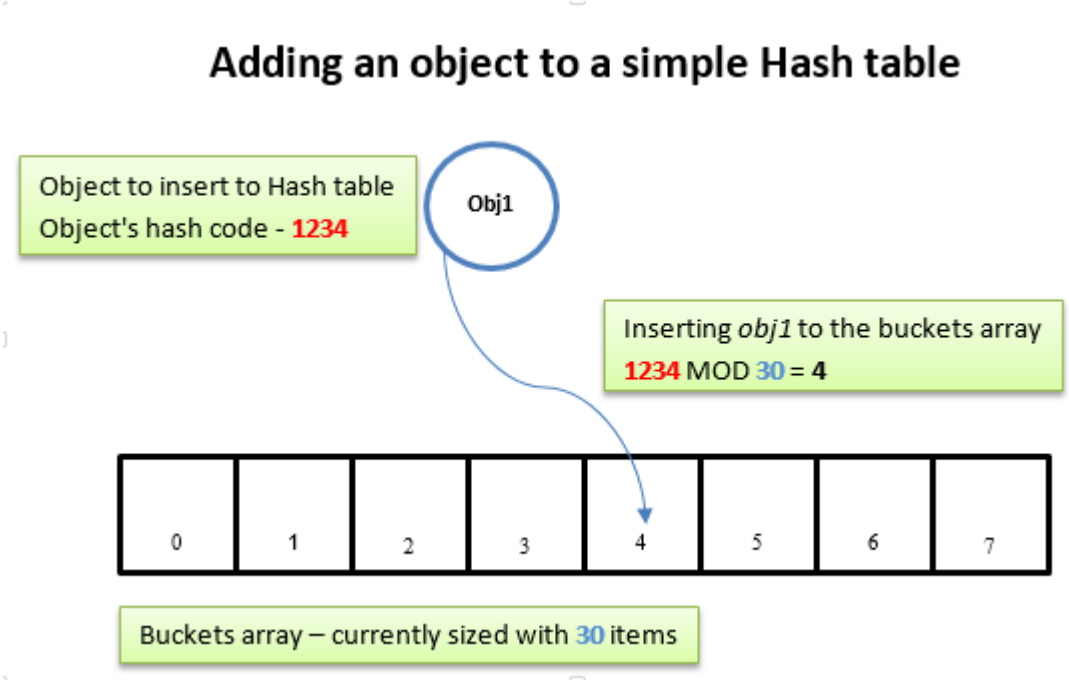
```
Hashtable hashtable = new Hashtable();
hashtable.Add(5, "peter");
hashtable.Add(3, 4);
object? x = hashtable[5];
```

HASHING

- Hashing ist der Prozess der Umwandlung eines Wertes in einen Hash-Wert
- Hash-Wert wird basierend auf einem Hashing-Algorithmus generiert

- Methode: GetHashCode()
- "Peter" => 1737667744

- Algorithmus: SHA256
- "Peter" => ea72c79594296e45b8c2a296644d988581f58cfac6601d122ed0a8bd7c02e8bf



LIST<T>

- Generische Collection
- Speichert eine Liste von Elementen des Typs T
- Effizient für sequenzielle Zugriffe und Iterationen
- Zugriff über den Index, z.B. myList[index]

```
List<int> list = new List<int>();
list.Add(2);
list.Add(54);
int x = list[2];
```

STACK<T>

- Generische Collection
- LIFO (Last-In-First-Out) Datenstruktur
- Zuletzt eingefügtes Element wird als erstes entfernt
- Dynamische Größte

```
Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
stack.Push(3);
int x = stack.Peek();
int y = stack.Pop();
```

QUEUE<T>

- Generische Collection
- FIFO (First-In-First-Out) Datenstruktur
- Zuerst eingefügtes Element wird als erstes entfernt
- Dynamische Größte

```
Queue<int> stack = new Queue<int>();
stack.Enqueue(1);
stack.Enqueue(2);
stack.Enqueue(3);
int x = stack.Peek();
int z = stack.Dequeue();
```

DICTIONARY<TKEY, TVALUE>

- Generische Collection
- Speichert Schlüssel-Wert-Paare
- Optimiert Suchvorgänge durch Hashing der Schlüssel
- Schlüssel: Eindeutig und nicht null
- Werte: Können null sein

```
var keyValuePairs = new Dictionary<int, string>();
keyValuePairs.Add(5, "Peter");
keyValuePairs.Add(4, "Franz");
bool x = keyValuePairs.ContainsKey(4);
string z = keyValuePairs[4];
keyValuePairs.Remove(4);
```