Professional Software Engineering

Andrea Carrara and Patrick Berggold

Hritik Singh and Mohab Hassaan – Tutors

Chair of Computational Modeling and Simulation

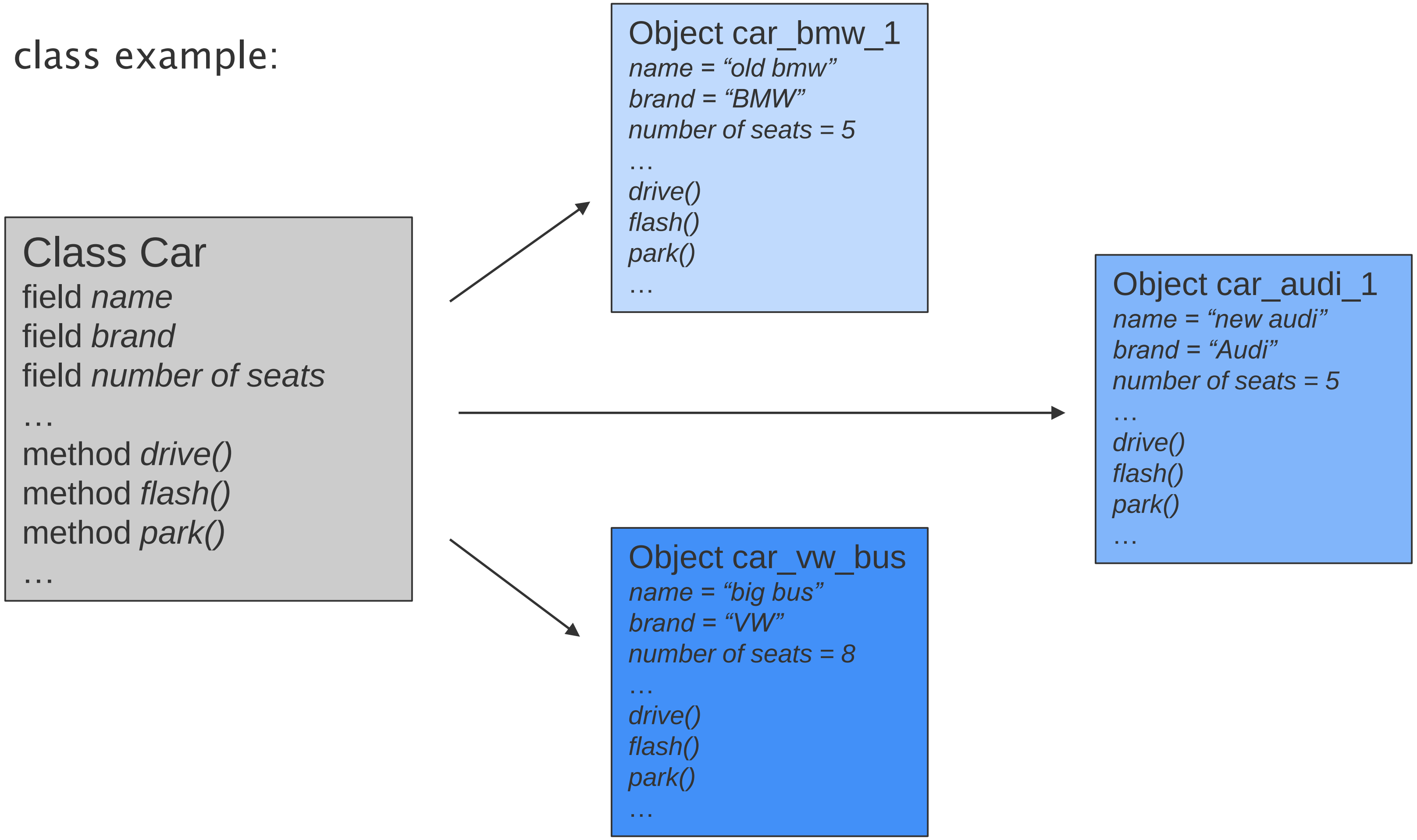
Schedule Lecture 4

- Generics
- Data structures
  - Array
  - List
  - Queue
  - Stack
  - Hashtables

RECAP OOP & INTERFACES

Class example

- A simple class example:



Class Car
field *name*
field *brand*
field *number of seats*
…
method *drive()*
method *flash()*
method *park()*
…

Object car_bmw_1
name = "old bmw"
brand = "BMW"
number of seats = 5
…
drive()
flash()
park()
…

Object car_audi_1
name = "new audi"
brand = "Audi"
number of seats = 5
…
drive()
flash()
park()
…

Object car_vw_bus
name = "big bus"
brand = "VW"
number of seats = 8
…
drive()
flash()
park()
…

Class Declaration and Object Instantiation in C#

```
// class declaration
class Vector2D {
    // fields = member variables
    public double x;
    public double y;

    // methods = member functions
    public double Norm() {
        double nrm = Math.Sqrt(x * x + y * y);
        return nrm;
    }
}
```

```
// object instantiation
Vector2D vec1 = new Vector2D();
vec1.x = 5;
vec2.y = 10;
Console.Write(vec1.Norm());
```

Virtual Methods – override

» A base method marked as virtual can be overridden by derived classes

» Keyword: override

» Used to provide a specialized implementation

» If casted to base again, will invoke the override the base method

» Not using override will issue a warning (not an error though)…

```
public class Figure {
    public virtual void Output() {
        Console.Write("Figure object");
    }
 }

 public class Circle : Figure {
    public override void Output() {
        Console.Write("Circle object");
    }
 }

 public class Rectangle : Figure {
    public override void Output() {
        Console.Write("Rectangle object");
    }
 }
```

Example for an Interface

```
public interface IRegularPolygon
{
    int NumberOfSides { get; set; }
    int SideLength { get; set; }


    double GetPerimeter();
    double GetArea();
}
```

- Start with an **I** (convention)
- All public!
- Properties
- No fields!
- Methods

Summary

- virtual: indicates that a method may be overridden by an inheritor
- override: overrides the functionality of a virtual method in a base class, providing different functionality.
- abstract: abstract methods must be implemented by the child and don’t contain a body. Abstract classes can only be inherited, never instantiated!
- interface: creates a “contract” for the derived class without any implementation. There can be many “contracts” for one single child class.

GENERICS

What are Generics

- Generic types are proxies for data types
- Allow you to define type-safe classes
- Only when you instantiate the generic class you have to specify the concrete type

Generics Implementation

» How to implement generics? Use the generic type parameter T!

```
// Class declaration and instantiation

class MyList<T> {
    // ...
}

class ListItem {
    // ...
}

// List of strings, integers and ListItems
MyList<string> list_of_words = new MyList<string>();
MyList<int> list_of_ints = new MyList<int>();
MyList<ListItem> list_of_items = new MyList<ListItem>();
```

```
// Method declaration and call

T1 SomeFunction<T1, T2>(T1 arg1, T2 arg2) {
    // returns variable of type T! (could also be void or
    // something else)
}
// Input parameters, e.g. of types int and string
int input_int = 0;
string input_str1 = "Hello there";
string input_str2 = "I like coding";

int result_as_int = SomeFunction(input_int, input_str2);
string result_as_str = SomeFunction(input_str1, input_str2);
```

Generics Constraints

- Constrain the datatypes via the `where` clause

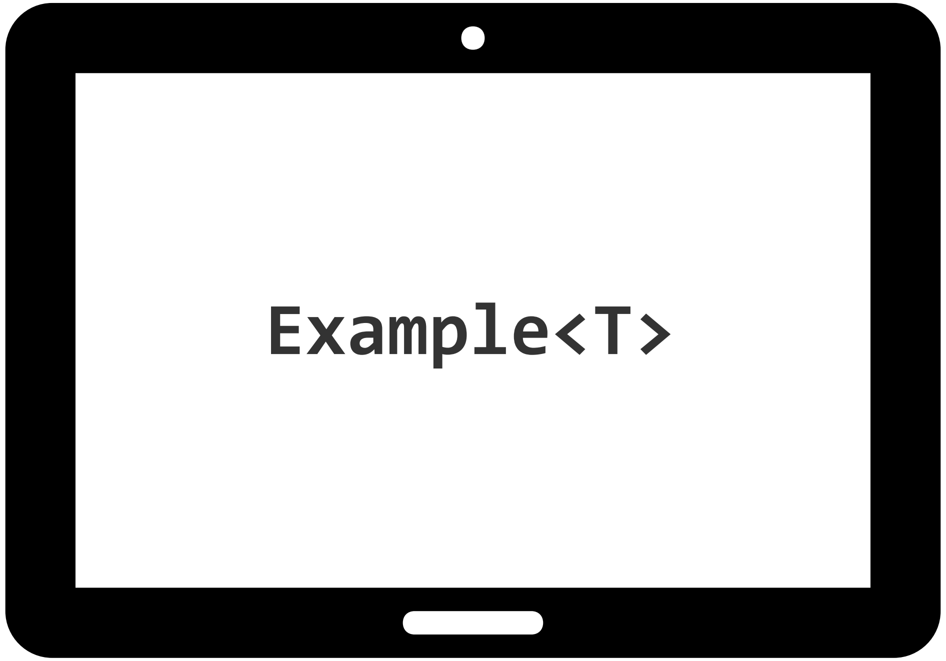
```
// Class declaration and instantiation

public class MyList<T> where T: ListItem {
    // ...
}

public class ListItem {
    // ...
}

// List of strings, integers and ListItems
MyList<string> list_of_words = new MyList<string>(); // throws an error!!!
MyList<ListItem> list_of_doubles = new MyList<ListItem>();
```

Constraining possible on one class, but several interfaces!!

There is a variety of constraints, more information here:

https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters

Generics Motivation & Implementation



Example<T>

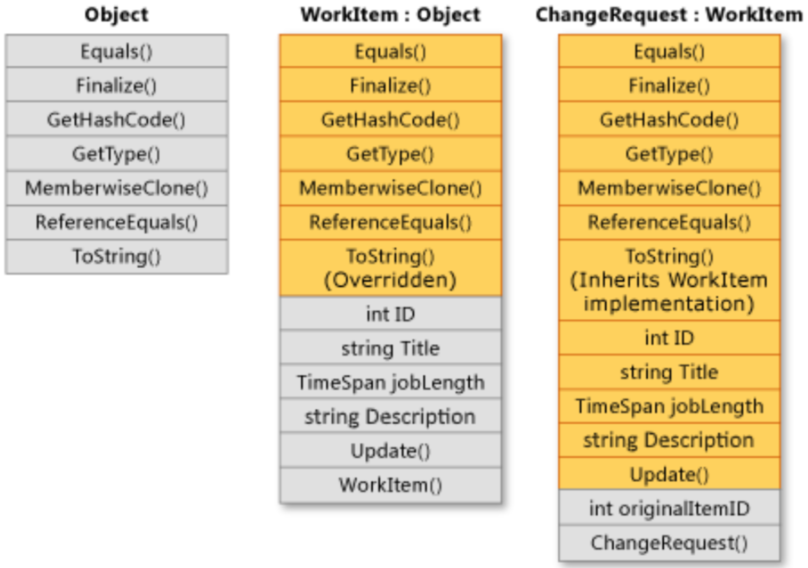
What are Generics

- Generic types are proxies for data types
- Allow you to define type-safe classes
- Only when you instantiate the generic class you have to specify the concrete type

- Basically no influence on performance! Significant speed improvement compared since boxing/unboxing is strictly avoided!
  - **E.g. accessing generic list (** `List<T>` **) is way faster than array list (** `ArrayList` **)**
- Comparable to C++ Templates and Java Generics

Boxing & Unboxing

- All classes inherit from the **System.Object** class
- Boxing refers to wrapping a value type inside a **System.Object** instance
  (this generates new memory allocation in another part of the memory)
- Unboxing refers to extracting a value type from an object or interface type



Object

Equals()
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals()
ToString()

WorkItem : Object

Equals()
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals()
ToString()
(Overridden)
int ID
string Title
TimeSpan jobLength
string Description
Update()
WorkItem()

ChangeRequest : WorkItem

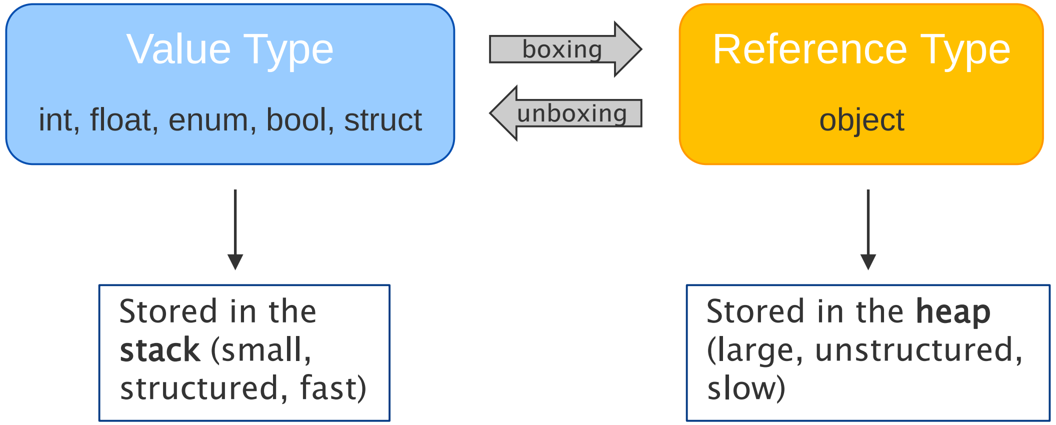
Equals()
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals()
ToString()
(Inherits WorkItem implementation)
int ID
string Title
TimeSpan jobLength
string Description
Update()
int originalItemID
ChangeRequest()

```
int someInt = 4;

object obj = someInt; // someInt is boxed (implicitly)

someInt = (int) obj; // someInt is unboxed (explicitly)
```

Boxing & Unboxing

- Flexibility, since value of any type can be treated as an object
- However, this process is computationally expensive and slow…

➡ Trade-off: flexibility vs. speed



Stored in the **stack** (small, structured, fast)

Stored in the **heap** (large, unstructured, slow)

Why is (un)boxing slow and expensive? Because we need to allocate new storage either on the stack or on the heap, and copy the data

- Which collections use object types?

  All non-generic collections (e.g. ArrayList, SortedList, Stack, Queue, HashTable)

For more details: https://www.c-sharpcorner.com/article/boxing-unboxing-in-c-sharp/

Arrays and Collections

DATA STRUCTURES

Arrays

- List of values of the same type

  (e.g. list of integer values, list of double values, list of boolean values)

- Length of list has to be defined when initializing
- Access to all values via one `int` variable (called the **index**)
- The first element is located at zero !

```
// all array initializations are possible
string[] array = new string[2];
string[] array = new string[] {"A", "B"};
string[] array = {"A", "B"};
string[] array = new[] {"A", "B"};
```

```
a[4] = 16;
```

index	0	1	2	3	4	5	6	7	8	9
array	0	1	4	9	16	25	36	49	64	81

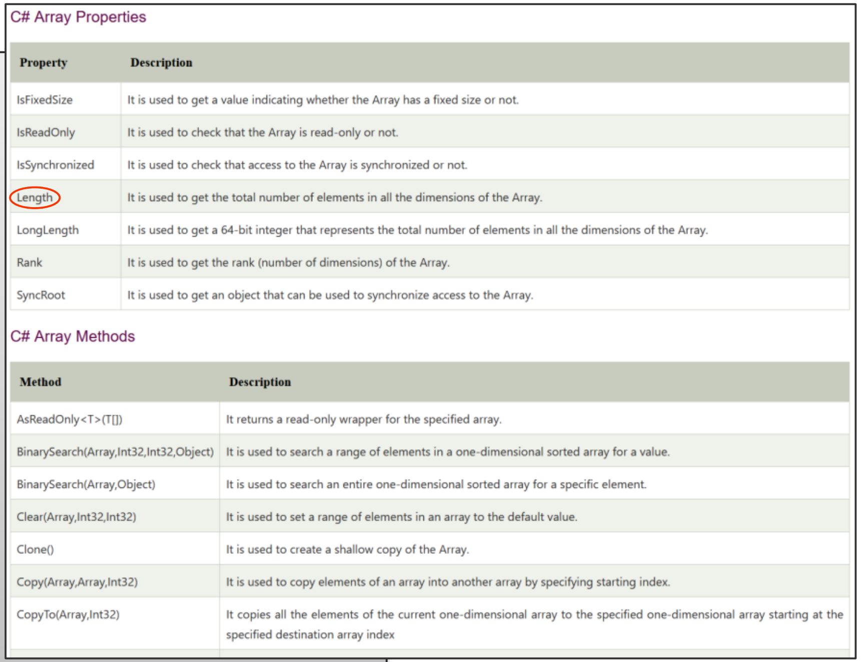
Array – Example

```
int[] a = new int[10];

for(int i = 0; i < 10; i++) {

    a[i] = i * i;

}

int x;

x = a[0];

x = a[3];

x = a[10]; // out of bounds

// Declare a two dimensional array

int[,] multiDimensionalArray1 = new int[2, 3];
```

index	0	1	2	3	4	5	6	7	8	9
array	0	1	4	9	16	25	36	49	64	81

Initialization & Length

```
char[] vowels = new char[5];

char[] vowels = new char[5] { 'a', 'e', 'i', 'o', 'u' };

char[] vowels = { 'a', 'e', 'i', 'o', 'u’ };


for(int i = 0; i < vowels.Length; i++) {
    Console.Write(vowels[i]);
} // print ‘aeiou’ in the console
```



C# Array Properties

Property	Description
IsFixedSize	It is used to get a value indicating whether the Array has a fixed size or not.
IsReadOnly	It is used to check that the Array is read-only or not.
IsSynchronized	It is used to check that access to the Array is synchronized or not.
Length	It is used to get the total number of elements in all the dimensions of the Array.
LongLength	It is used to get a 64-bit integer that represents the total number of elements in all the dimensions of the Array.
Rank	It is used to get the rank (number of dimensions) of the Array.
SyncRoot	It is used to get an object that can be used to synchronize access to the Array.

C# Array Methods

Method	Description
AsReadOnly<T>(T[])	It returns a read-only wrapper for the specified array.
BinarySearch(Array,Int32,Int32,Object)	It is used to search a range of elements in a one-dimensional sorted array for a value.
BinarySearch(Array,Object)	It is used to search an entire one-dimensional sorted array for a specific element.
Clear(Array,Int32,Int32)	It is used to set a range of elements in an array to the default value.
Clone()	It is used to create a shallow copy of the Array.
Copy(Array,Array,Int32)	It is used to copy elements of an array into another array by specifying starting index.
CopyTo(Array,Int32)	It copies all the elements of the current one-dimensional array to the specified one-dimensional array starting at the specified destination array index

https://www.javatpoint.com/c-sharp-array-class

char Array vs. string

```
char[] vowels_c = { 'a', 'e', 'i', 'o', 'u’ }; // read and write
string vowels_str = "aeiou"; // read-only because of immutability


// char manipulation
vowels_c[3] = '0'; // changes the array to { 'a', 'e', 'i', '0', 'u’ }
// string manipulation
vowels_str[3] = "0"; // throws an error


// string immutability is bypassed by internally creating an edited copy of the string
vowels_str += " edited"; // redefined 'aeiou edited’, same as vowels_str = vowels_str + " edited"
vowels_str = vowels_str.Remove(2) // built-in string method Remove() removes all chars from 3rd char
```

These are not similar!!!

You can re-assign a new value to a variable, but you cannot edit an existing immutable object

Applying Interfaces

COLLECTIONS

Overview

- Collections are classes for data storage and organization
- Arrays are most useful for creating and working with a fixed number of objects.

```
// general (non-)generic instantiation like class
Collection<data type> name = new Collection <data type>();
Collection name = new Collection();
```

- Not always but most of the time:
  - **dynamic size (growing and shrinking)**
- Namespace:
  - **System.Collections (.Generic)**

```
// to the top of the file
using System.Collections;
using System.Collections.Generic;
```

- Capabilities defined by the interfaces they implement

Example Collections

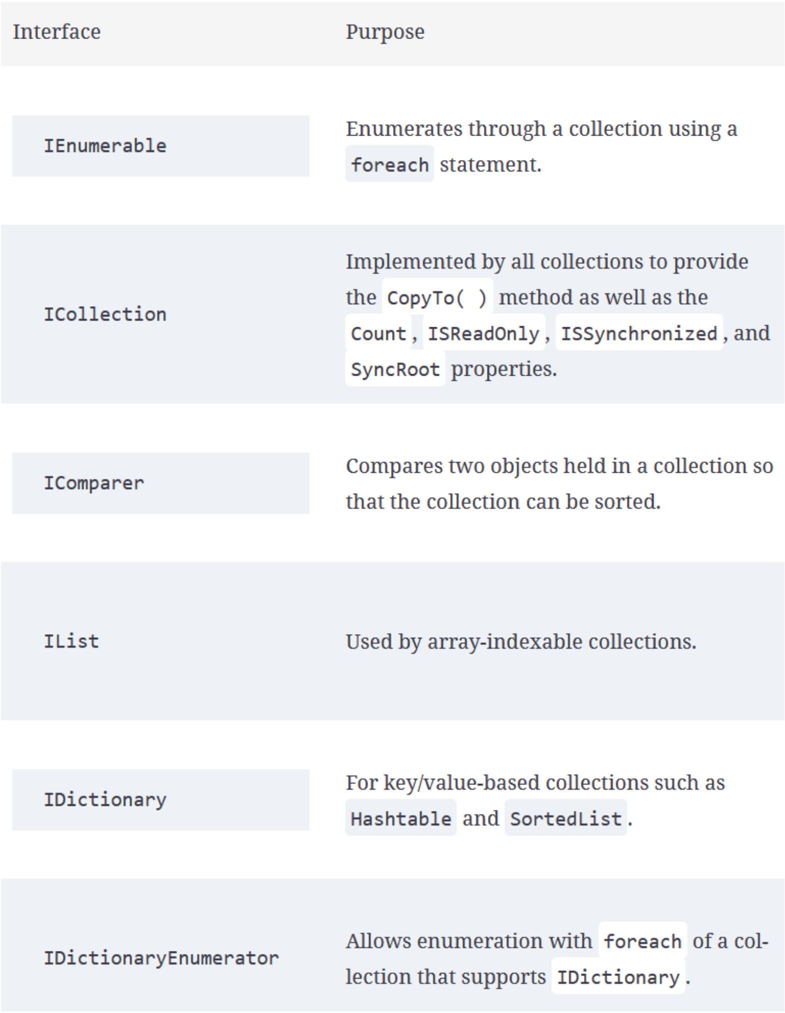
- List<T>
- ArrayList
- SortedList<TKey, TValue>
- HashTable
- Queue<T>, Queue
- Stack<T>, Stack
- Dictonary<TKey, TValue>
- ObsevervableCollection<T>

All non-generic collections store objects!

Collections implement interfaces

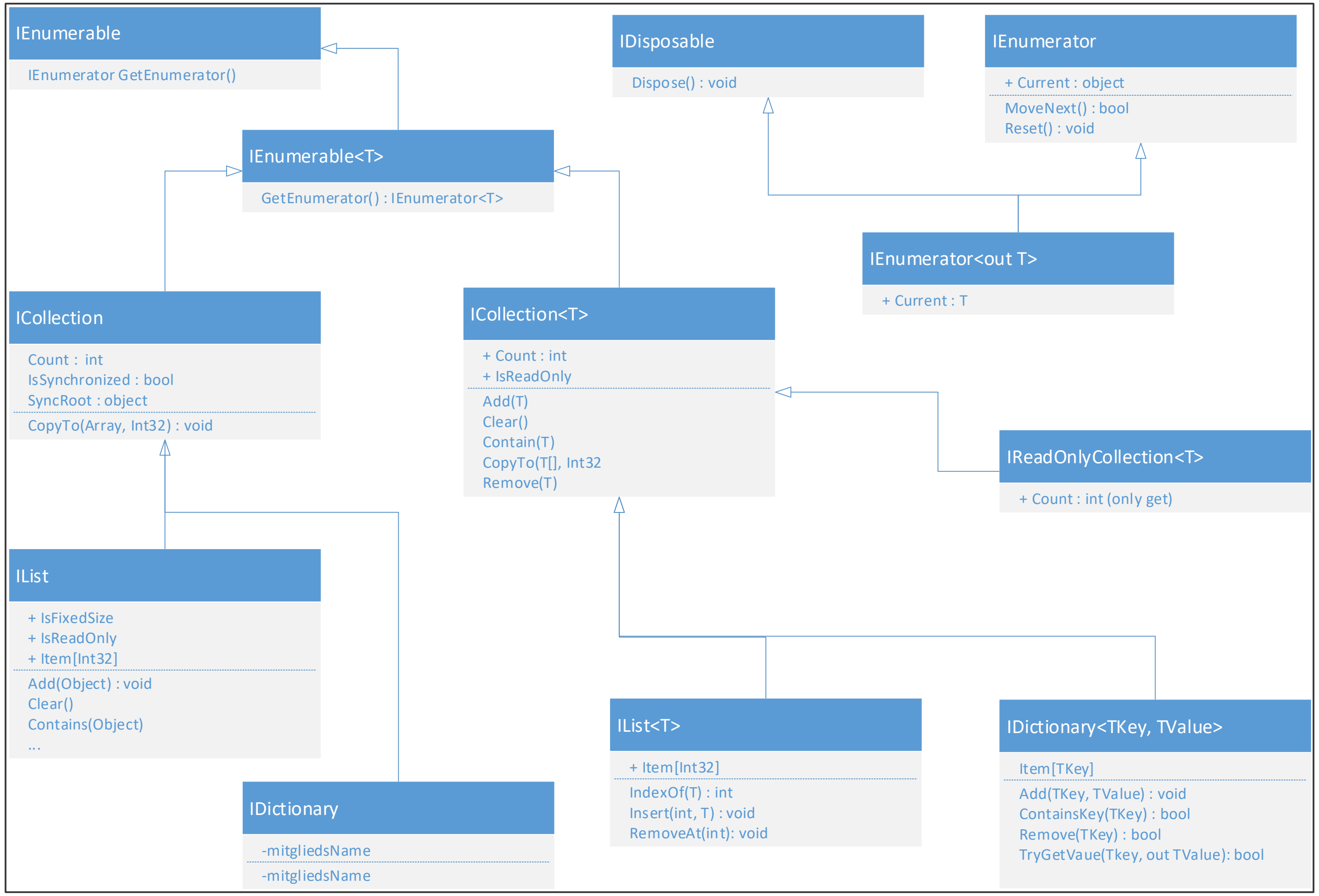
- » To work with collections, they implement some useful functionalities from interfaces
- » The .NET Framework provides standard interfaces for enumerating, comparing, and creating collections
- » Example:
  - – `IEnumerable`
    - → `GetEnumerator()`
    - → Makes `foreach()` possible!
  - – `IList`
    - → Make indexing possible (`someArray[3]`)

From *Programming C#*, Jessy Liberty, table 9-2



Interface	Purpose
`IEnumerable`	Enumerates through a collection using a `foreach` statement.
`ICollection`	Implemented by all collections to provide the `CopyTo( )` method as well as the `Count`, `ISReadOnly`, `ISSynchronized`, and `SyncRoot` properties.
`IComparer`	Compares two objects held in a collection so that the collection can be sorted.
`IList`	Used by array-indexable collections.
`IDictionary`	For key/value-based collections such as `Hashtable` and `SortedList`.
`IDictionaryEnumerator`	Allows enumeration with `foreach` of a collection that supports `IDictionary`.

Interface hierarchy



IEnumerable

IEnumerator GetEnumerator()

IDisposable

Dispose() : void

IEnumerator

+ Current : object

MoveNext() : bool

Reset() : void

IEnumerable<T>

GetEnumerator() : IEnumerator<T>

IEnumerator<out T>

+ Current : T

ICollection

Count : int

IsSynchronized : bool

SyncRoot : object

CopyTo(Array, Int32) : void

ICollection<T>

+ Count : int

+ IsReadOnly

Add(T)

Clear()

Contain(T)

CopyTo(T[], Int32

Remove(T)

IReadOnlyCollection<T>

+ Count : int (only get)

IList

+ IsFixedSize

+ IsReadOnly

+ Item[Int32]

Add(Object) : void

Clear()

Contains(Object)

...

IList<T>

+ Item[Int32]

IndexOf(T) : int

Insert(int, T) : void

RemoveAt(int): void

IDictionary<TKey, TValue>

Item[TKey]

Add(TKey, TValue) : void

ContainsKey(TKey) : bool

Remove(TKey) : bool

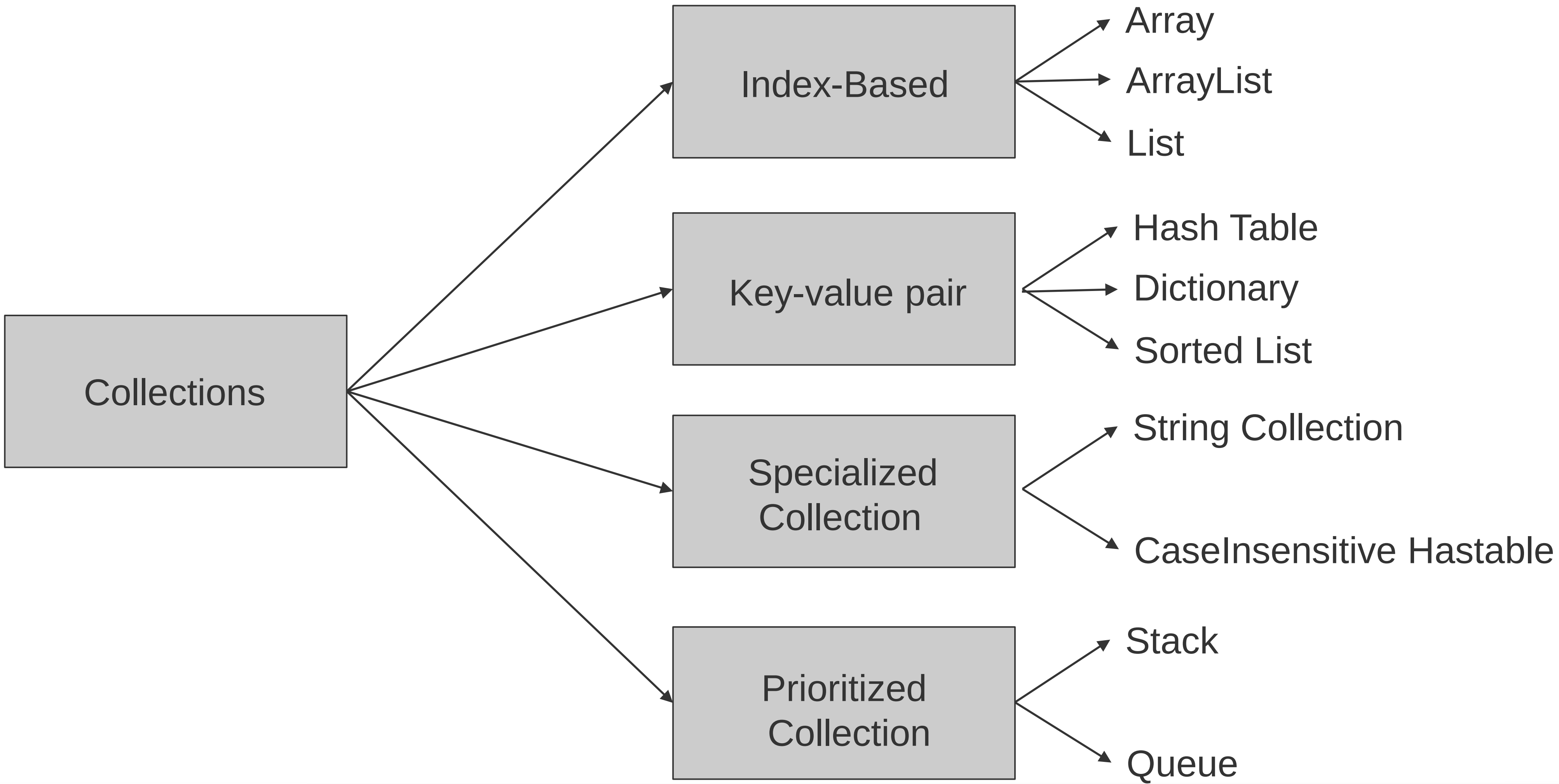
TryGetVaue(Tkey, out TValue): bool

IDictionary

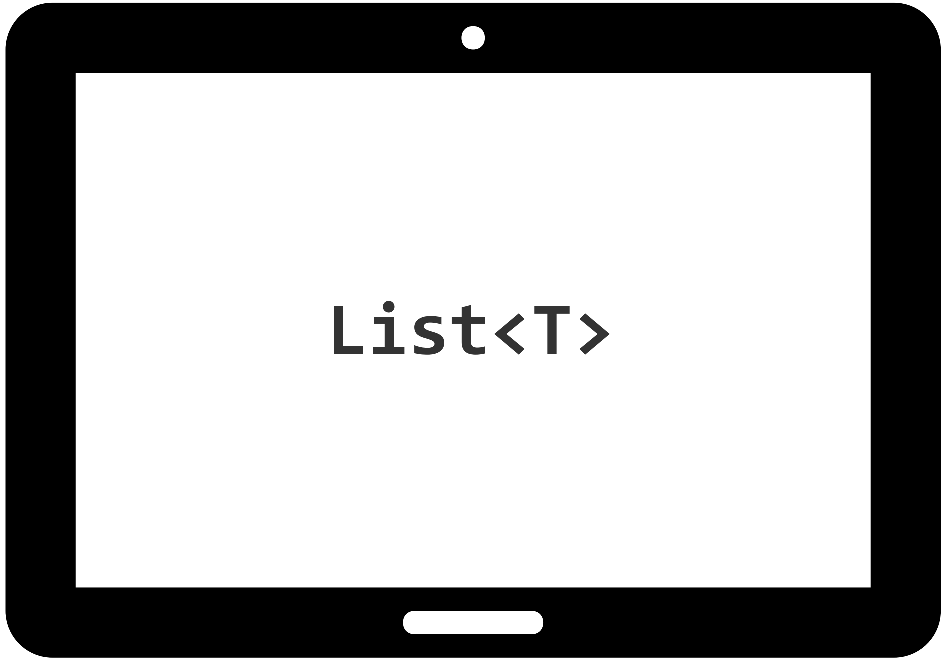
-mitgliedsName

-mitgliedsName

Collections



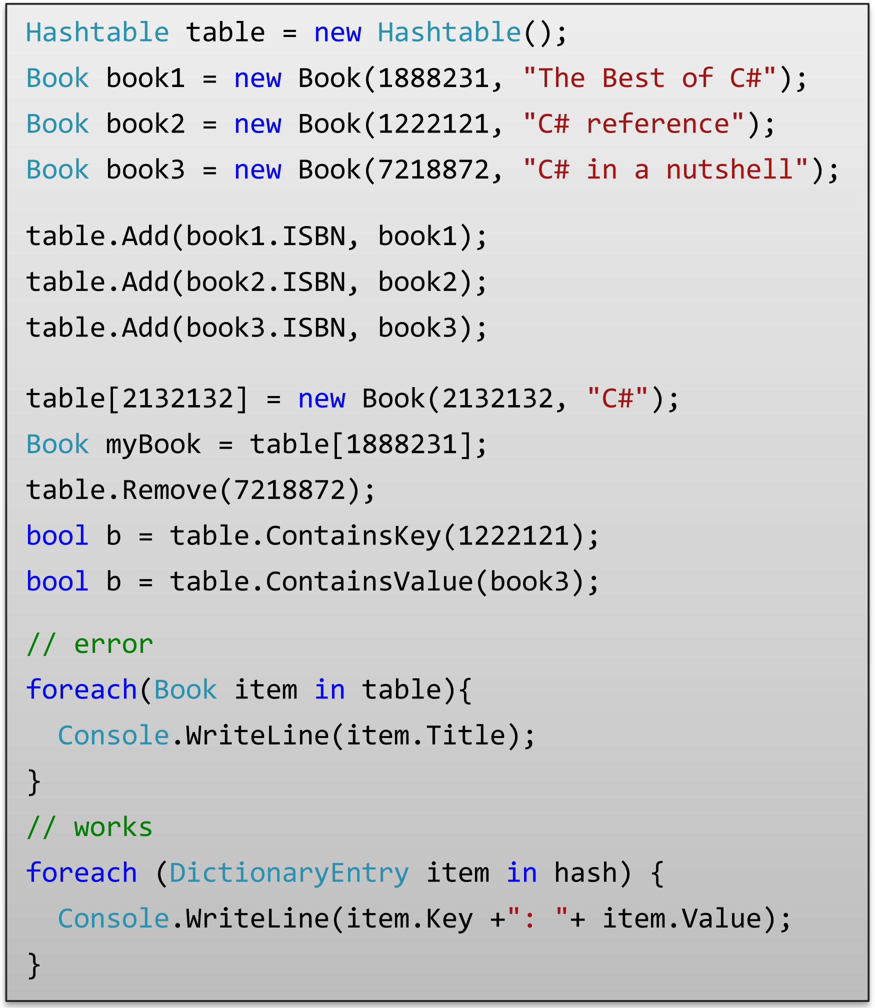
List Implementation



List<T>

HashTable

- Stores key-value pairs
- Allows look-up by key: HashTable[key] = value
- Non-generic
- Implements
  - `IDictionary, ICollection, IEnumerable, ISerializable, ICloneable IDeserializationCallback`
- To iterate, use `DictionaryEntry`
- Called a HashTable because it creates a unique hash code from the key and sorts elements according to that hash code

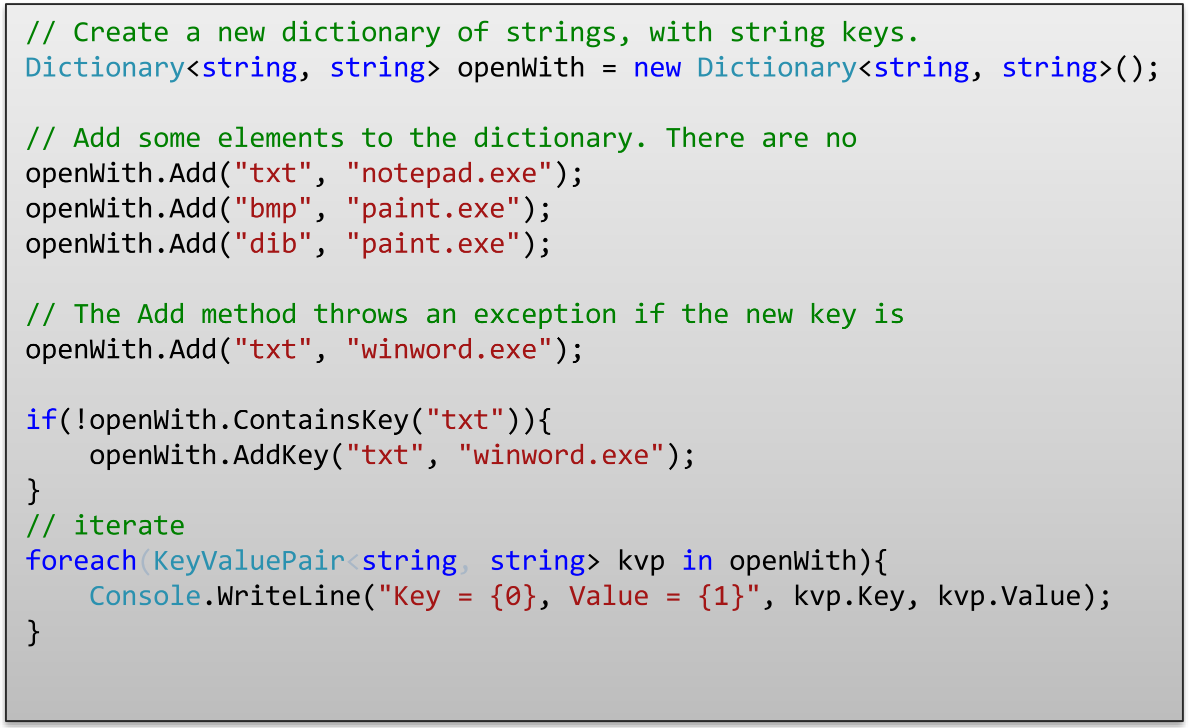


```
Hashtable table = new Hashtable();
Book book1 = new Book(1888231, "The Best of C#");
Book book2 = new Book(1222121, "C# reference");
Book book3 = new Book(7218872, "C# in a nutshell");

table.Add(book1.ISBN, book1);
table.Add(book2.ISBN, book2);
table.Add(book3.ISBN, book3);

table[2132132] = new Book(2132132, "C#");
Book myBook = table[1888231];
table.Remove(7218872);
bool b = table.ContainsKey(1222121);
bool b = table.ContainsValue(book3);

// error
foreach(Book item in table){
  Console.WriteLine(item.Title);
}
// works
foreach (DictionaryEntry item in hash) {
  Console.WriteLine(item.Key +": "+ item.Value);
}
```

Dictionary

- » Stores key-value pairs
- » Allows look-up by key: Dictionary[key] = value
- » Generic

- » To iterate, use KeyValuePair
- » Key is also hashed (it is basically a HashTable), but the key are ordered according to insertion

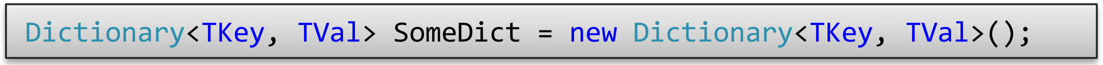


```
// Create a new dictionary of strings, with string keys.
Dictionary<string, string> openWith = new Dictionary<string, string>();

// Add some elements to the dictionary. There are no
openWith.Add("txt", "notepad.exe");
openWith.Add("bmp", "paint.exe");
openWith.Add("dib", "paint.exe");

// The Add method throws an exception if the new key is
openWith.Add("txt", "winword.exe");

if(!openWith.ContainsKey("txt")){
    openWith.AddKey("txt", "winword.exe");
}
// iterate
foreach(KeyValuePair<string, string> kvp in openWith){
    Console.WriteLine("Key = {0}, Value = {1}", kvp.Key, kvp.Value);
}
```

Dictionary vs HashTable

Dictionary:

- Generic, type-safe, in namespace **System.Collections.Generic**

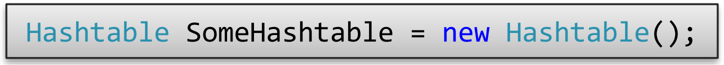


```
Dictionary<TKey, TVal> SomeDict = new Dictionary<TKey, TVal>();
```

- Enumerated item of type `KeyValuePair`
- Request to non-existing key throws exception
- Maintains insertion order
- Type-safety avoids boxing & unboxing

HashTable:

- Non-generic, not type-safe, in namespace **System.Collections**

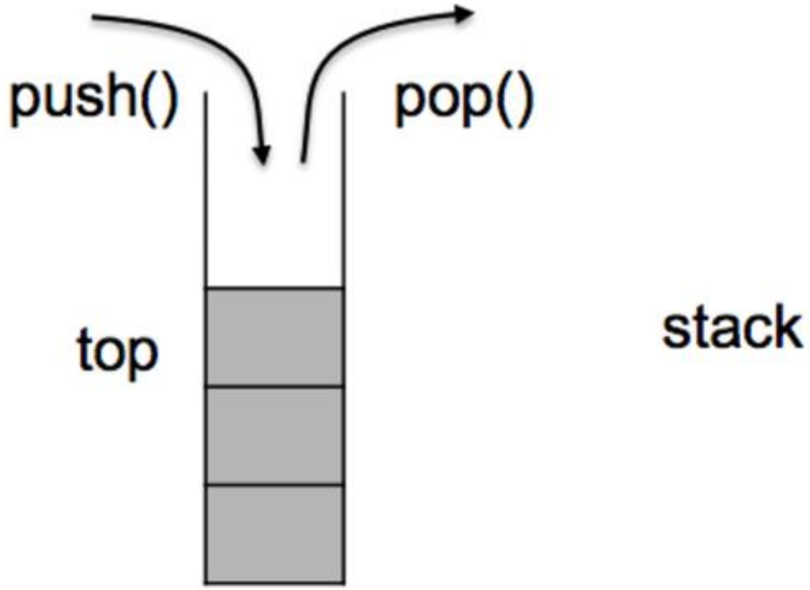


```
Hashtable SomeHashtable = new Hashtable();
```

- Enumerated item of type `DictionaryEntry`
- Request to non-existing key returns `null`
- Does not maintain insertion order
- Boxing & unboxing leads to lower speed

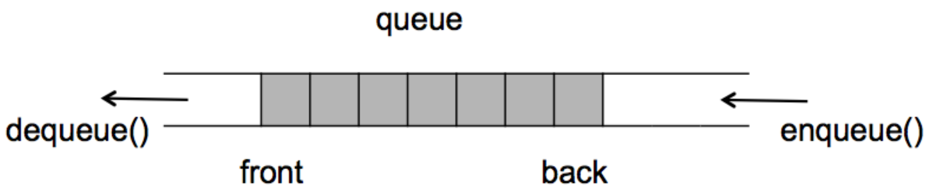
Stack, Stack<T>

- May be generic or not
- Special collection to insert/remove objects only at the top (LIFO)
- Implements
  - **IEnumerable<T>, IEnumerable, ICollection, IReadOnlyCollection<T>**
- Methods:
  - **Peek()**
  - **Pop()**
  - **Push()**



Queue, Queue<T>

- May be generic or not
- First In – First Out (FIFO)
- Useful for storing messages in the order they were received
- Implements:
  - `IEnumerable<T>, IEnumerable, ICollection, IReadOnlyCollection<T>`
- Methods:
  - `Dequeue(), Enqueue(), Peek()`



Summary Data Structures

- Generics are type-safe
- Arrays store Data of the same Datatype (also Objects)
  - They have to be initialized at compile-time (including their size)
- Collections are predefined Classes to handle Objects
  - Arrays are static whereas Collections are dynamic structures, hence the memory used for computation can be allocated dynamically
  - Collections are using interfaces
  - Hash-based systems (e.g. Dictionary, HashTable) ensure the objects are accessible by key
  - Stacks and Queues ensure the stored objects are processed in a certain order

THANK YOU!