Lecture 2. Collections and generics

Programming II

School of Business Informatics
Autumn 2016

(: 6 stages of debugging:

1. That can't happen 2. That doesn't happen on my computer 3. That shouldn't happen 4. Why does that happen? 5. Oh, I see. 6. How did that ever work? :)

Collections

- Collections are required to create and manage groups of related objects
- .NET has about 15! classes representing different types of collections

Collections

- Collections are required to create and manage groups of related objects
- .NET has about 15! classes representing different types of collections

Most commonly used collections - array, string, List, LinkedList, Queue, Stack, Dictionary.

Collections

- Collections are required to create and manage groups of related objects
- .NET has about 15! classes representing different types of collections

Most commonly used collections - array, string, List, LinkedList, Queue, Stack, Dictionary.

Why so many?

Common features

- Collections are reference types (items are stored on the heap)
- Standard collections (except strings) are mutable (can be changed after initialization)
- All modern collection classes are strongly typed. Loosely typed classes, e.g. ArrayList, HashTable, are included for backward compatibility

Difference between collections

- Internal organization
- Efficiency of different operations
- Allocated memory
- Presence of notifiers (important for automatic updates of the UI)
- Thread-safety (will be covered later in the course)

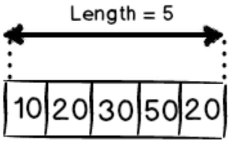
Main operations on containers

- Get an element by index / key
- Add element to the back / to the front / at arbitrary index
- Remove element from the back / from the front / from an arbitrary index
- Iterate through all elements

Array

- ✓ Single block of memory allocated for all elements
- ✓ Elements are equal in size
- ✓ Efficient element access by index
- ✗ Dynamic resizing

```
int[] array = new int[] {10, 20, 30, 50, 20};
```



string vs char[]

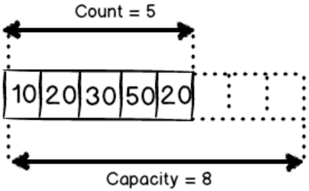
- A string is **immutable**, i.e. after initialization its contents cannot be changed in place. All operations on strings - Concat, ToLower, ToUpper, Remove, Replace, Trim and others - create a new string in memory preserving the old one
- A char[] or List<char> is **mutable**: individual characters can be changed after initialization

```
class String
{
    // string -> char[]
    public String(char[] value);
    // char[] -> string
    public char[] ToCharArray();
}
```

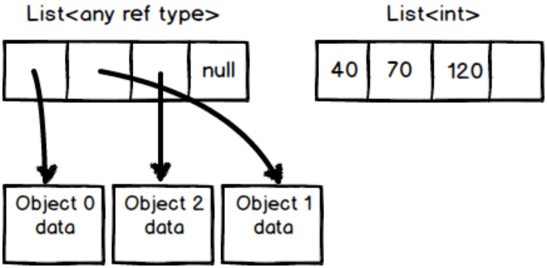
List

- ✓ Most commonly used container
- ✓ Internally organized as an array
- ✓ Additional logic added to dynamically resize the internal array when no free space is left
- ✗ Efficient insertion to the front /removal from the front

```
List<int> list = new List<int> {10, 20, 30, 50,
    20};
```



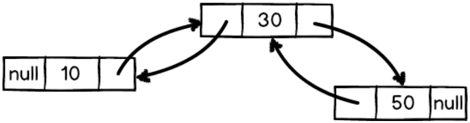
Storing values and references



Linked list

- ✓ Elements are stored in non-sequential blocks of memory
- ✓ Efficient insertion to the front / removal from the front
- ✗ Access to elements by index

```
LinkedList<int> linkedList = new LinkedList<int
    >();
linkedList.AddLast(30);
linkedList.AddLast(50);
linkedList.AddFirst(10);
```



Associative container

- An associative container (AC) is formed by key-value pairs (for each key an AC stores the value associated with it) and enables quick retrieval of a value by its key
- Several .NET classes implement an associative container, the most common is:

```
class Dictionary<TKey, TValue>
```

- A Dictionary is also very efficient at inserting and deleting key-value pairs

Dictionary examples

- A user enters a month name and the program outputs the number of days in the corresponding month.
- Association between a file extension and a default program to open such files.
- Important events that happened on a particular day in history.

Dictionary examples

- A user enters a month name and the program outputs the number of days in the corresponding month. Name of a month - Number of days
- Association between a file extension and a default program to open such files.
- Important events that happened on a particular day in history.

Key - Value

Dictionary examples

- A user enters a month name and the program outputs the number of days in the corresponding month. Name of a month - Number of days
- Association between a file extension and a default program to open such files. Extension - Program name/path
- Important events that happened on a particular day in history.

Key - Value

Dictionary examples

- A user enters a month name and the program outputs the number of days in the corresponding month. Name of a month - Number of days
- Association between a file extension and a default program to open such files. Extension - Program name/path
- Important events that happened on a particular day in history. Date - List of events

Key - Value

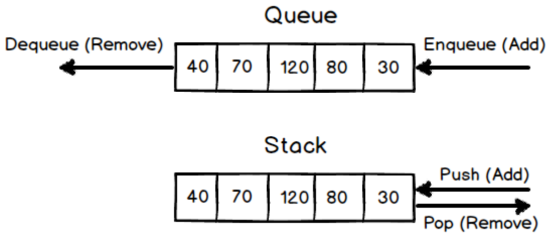
Dictionary rules

- Each key appears only once in a dictionary. Values can be repeated.
- The dictionary **key** has to be of an **immutable** type (int, double, char, string, DateTime and others)
- The order, in which key-value pairs are stored and then retrieved from a Dictionary (e.g. in a foreach loop) cannot be easily predicted and can be considered as random
- SortedDictionary and SortedList are two examples of ACs, which allow to retrieve keys in a sorted order.

An article on internal dictionary structure

Queue and Stack

Array or linked list based collections with a special add-remove logic:



Queue and Stack usage scenarios

- Intermediate buffer between components (both hardware and software) operating at different speeds
- Search algorithms
- Backtracking
- Expression parsing

Problem of similar methods

Consider the following method that exchanges values of two integer variables:

```
static void Swap(ref int num1, ref int num2)
{
    int temp = num1;
    num1 = num2;
    num2 = temp;
}
```

Problem of similar methods

Consider the following method that exchanges values of two integer variables:

```
static void Swap(ref int num1, ref int num2)
{
    int temp = num1;
    num1 = num2;
    num2 = temp;
}
```

What if we need to exchange two “double” values?

Generic version of Swap

```
static void Swap<T>(ref T num1, ref T num2)
{
    T temp = num1;
    num1 = num2;
    num2 = temp;
}
```

Swap can now be used to interchange variables of any type.

Generic class example

Generic principles can also be applied to classes.

```
class GenericItem<T>
{
        T Value { get; set; }
        string Comment { get; set; }
}
```

In the example above T can be used for any member of the class.

Generics in .NET

Generic classes are widely used in .NET Framework (first appeared in .NET v2.0)

- Collections
- Anonymous delegates and lambda expressions
- LINQ
- Entity framework
- and many other standard features

Why use generics?

Generics offer a number of advantages:

- Type safety is ensured at compile time
- Boxing does not occur in case T is a value type

Universal programming principle: Compile time errors are preferable to runtime errors.

Restrictions of generic classes

```
static T Sum<T>(T[] array)
{
    T sum = 0;
    foreach (var item in array)
        sum += item;
    return sum;
}
```

In the example above:

- Cannot assign a variable to 0
- Cannot add values

Partial solution to the problem: constraints on generic classes

Constraints on generic classes

```
public class GenericClass<T> where <constraints>
{
}
```

- where T : class
- where T : struct
- where T : new()
- where T : <Name of Base class>
- where T : <Name of Interface>