

CHAPTER 7

C#

COLLECTIONS AND GENERIC

Contents

1. Array & String
2. Collection
   - ArrayList
3. Generic
   - Stack and Queue
   - List, SortedList and LinkedList
   - Dictionary
   - User generic classes

C# Array

- An array stores a fixed-size **sequential collection of elements** of the same type.
  - An array is collection of variables of the same type stored at **contiguous memory** locations.
- In **C#** the allocation of memory for the arrays is done dynamically.
  - And arrays are kind of objects, therefore it is easy to find their size using the predefined functions.
- C# array is an object of base

Array Declaration

- **Syntax :**
- < Data Type > [ ] < Name_Array >
  - **Here,**
    - < Data Type > : It define the element type of the array.
      [ ] : It define the size of the array.
      < Name_Array > : It is the Name of array.
  - Example
    - int[] x; // can store int values
    - string[] s; // can store string values
    - double[] d; // can store double values
    - Student[] stud1; // can store instances of Student class which is custom class

Array Initialization

- As said earlier, an array is a reference type so the **new** keyword used to create an instance of the array. We can assign initialize individual array elements, with the help of the index.
- **Syntax :**
  - type [ ] < Name_Array > = new < datatype > [size];
- **Example**
  - int[] intArray1 = new int[5];
  - int[] intArray2 = new int[5]{1, 2, 3, 4, 5};
  - int[] intArray3 = {1, 2, 3, 4, 5};

Array Length

- To find out how many elements an array has, use the **Length** property:
- **Example**
  - string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
  - Console.WriteLine(cars.**Length**); // Outputs 4

Sort Arrays

- There are many array methods available, for example Sort(), which sorts an array alphabetically or in an ascending order:
- Example
  - string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
  - Array.**Sort**(cars);

- **System.Linq** Namespace
  - Other useful array methods, such as **Min**, **Max**, and **Sum**, can be found in the System.Linq namespace:
  - Example
    - int[] myNumbers = {5, 1, 8, 9};
    - myNumbers.Max(); // returns the largest value
    - myNumbers.Min(); // returns the smallest value
    - myNumbers.Sum(); // returns the sum of elements

- Concept & Description
  - **Multi-dimensional arrays**
    - C# supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
  - **Jagged arrays**
    - C# supports Jagged arrays, whose elements are arrays. It means “**array of arrays**”
  - **Passing arrays to functions**
    - You can pass to the function a pointer to an array by specifying the array's name without an index.

Exercise 1:

- Write a program in C# Sharp to count the frequency of each element of an array.
  - Test Data :

Input the number of elements to be stored in the array :3

Input 3 elements in the array :

element - 0 : 25

element - 1 : 12

element - 2 : 43

  - *Expected Output* :

Frequency of all elements of array :

25 occurs 1 times

12 occurs 1 times

43 occurs 1 times

C# String

- Strings are used for storing text.
  - A string variable contains a **collection of characters** surrounded by double quotes.
    - **Example**
      - string greeting = "Hello";
  - **String Length**
    - The length of a string can be found with the Length property:
      - greeting.**Length**;
  - **ToUpper()** and **ToLower()**
    - Returns a copy of the string converted to uppercase or lowercase
      - greeting .ToUpper(); // Outputs "HELLO“
      - greeting .ToLower(); // Outputs "hello"

- **Access Strings**
  - You can access the characters in a string by referring to its index number inside **square brackets []**.
    - Example
      - gretting[0] // return ‘H’
  - String indexes start with 0: [0] is the first character. [1] is the second character, etc.
  - You can also find the index position of a specific character in a string, by using the **IndexOf**() method:
    - gretting.IndexOf("e") // return 1

- String **Concatenation**
  - The **+ operator** can be used between strings to combine them. This is called **concatenation**:
  - You can also use the **string.Concat**() method to concatenate two strings:
- **Special Characters**
  - Because strings must be written within quotes, C# will misunderstand this string, and generate an error:
    - string txt = "We are the so-called "Vikings" from the north.";
  - The solution to avoid this problem, is to use the **backslash escape character**.
    - The backslash (\) escape character turns special characters into string characters: (\' , \" , \\)

Exercise 2:

- Write a program in C# Sharp to count the total number of words in a string.
  - Test Data :

    Input the string : This is the second exercise
  - *Expected Output* :

    Total number of words in the string is : 5
- **Note**
  - Check whether the current character is **white space** or **new line** or **tab character**

Collection

- The .NET framework provides specialized classes for data storage and retrieval. In the previous slides, we have described arrays.
- Collections are enhancement to the arrays.
- There are three distinct collection types in C#:
  1. Standard
  2. Generic
  3. Concurrent

1. The **standard collections** are found under the **System.Collections**.
   - They **do not store** elements as specifically typed objects, but as objects of type Object. Standard collections include **ArrayList**, **Hashtable**, **Queue**, and **Stack**.
2. The **generic collections** are found under S**ystem.Collections.Generic**.
   - Generic collections are more flexible and are the preferred way to work with data.

- Generics enhance code reuse, type safety, and performance. The generic collections include **Stack<T>, Dictionary<T,T>, List<T>, Queue< T >, SortedList<T>,** and **Stack<T>.**

3. **Concurrent collections** include **BlockingCollection <T>**, **ConcurrentDictionary<T, T>, ConcurrentQueue<T>**, and **ConcurrentStack<T>**.

C# ArrayList

- **ArrayList** is a collection from a standard *System.Collections* namespace.
  - It is a **dynamic array** (An ArrayList automatically expands as data is added)
  - It provides **random access** to its elements.
  - Unlike arrays, an ArrayList can hold **data of multiple data types**.

- Elements in the ArrayList are **accessed via an integer index**. Indexes are zero based.
- Indexing of elements and insertion & deletion at the end of the ArrayList takes constant time.
  - (Inserting or deleting an element in the middle of the dynamic array is more costly. It takes linear time)

Example










```
using System.Collections;
namespace ConsoleApplication_Collection1
{
    1 reference
    class Empty {}
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            var data = new ArrayList();

            data.Add("C#");
            data.Add(344);
            data.Add(55);
            data.Add(new Empty());
            data.Remove(55);

            foreach (object el in data)
            {
                Console.WriteLine(el);
            }
            Console.ReadKey();
        }
    }
}
```

```
C#
344
ConsoleApplication_Collection1.Empty
```

1. using System.Collections;
   - In order to work with ArrayList collection, we need to use the System.Collections namespace.
2. var data = new ArrayList();
   - An ArrayList collection is created
3. data.Add(“C#");
   - We add four elements to the array with the **Add**() method.
4. data.Remove(55);
   - We remove one element with the **Remove**() method

Other type of Standard (Non-Generic)Collections

- Class **Hashtable**

  // create a new hash table

  Hashtable table = new **Hashtable()**;

- Class **Queue**

  // create an empty Queue

  Queue queue= new **Queue()**;

- Class **Stack**

  // create an empty Stack

  Stack stack = new **Stack()**;

Generic Collections

- Generics introduce to the .NET Framework the concept of type parameters.
  - This makes it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated.
  - The .NET Framework class library contains several new generic collection classes in the **System.Collections.Generic** namespace.

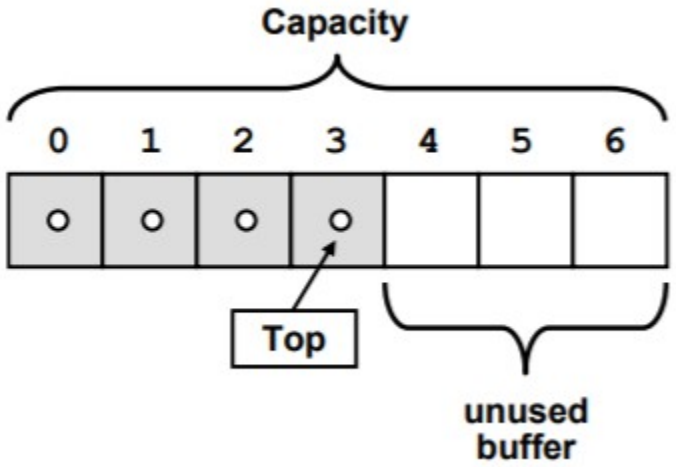
Generic type	Description
Stack<T>	Represents a variable size last-in first-out collection of instances
Queue<T>	Represents a firs-in, first out collection of objects
List<T>	Represents a strongly typed list of objects
SortedDictionary<Tkey, TValue>	Represents a collection of key/value pairs that are sorted on the key
LinkedList<T>	Represents a doubly linked list
Dictionary<Tkey, TValue>	Represents a collection of keys and values

C# Stacks

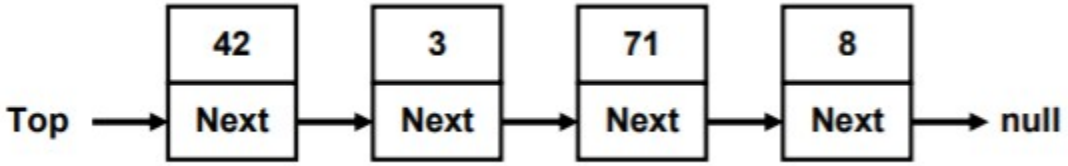
- A ***stack*** is a **Last-In-First-Out (LIFO)** data structure.
  - The **last** element **added** to the stack will be the **first** one to be **removed**.
  - We could **add elements on the top** and remove the element, which has been added last, but no the previous ones (the ones that are below it).
  - The stack is used internally by the **.NET virtual machine** (CLR) **for keeping the variables of the program** and the **parameters of the called methods** (it is called program execution stack).

- Stack provides 3 major operations:
  - **push** - add an element at the top of the stack
  - **pop** - take the last added element from the top of the stack
  - **peek** - get the element form the top of the stack without removing it.
- The data structure stack can also have different implementations,
  - **Static** : using Array implementation
  - **Dynamic :** Linked list implementation

- Static Stack



- Dynamic Stack



Stack<T> class

- In C# we could use the standard implementation of the class in .NET Framework **System.Collections.Generics.Stack**.
- The Stack Class – Basic Operations
  - **Push(T)** - adds a new element on the top of the stack
  - **Pop()** - returns the highest element and removes it from the stack
  - **Peek()** - returns the highest element without removing it
  - **Count** - returns the count of elements in the stack
  - **Clear()** - retrieves all elements from the stack
  - **Contains(T)** - check whether the stack contains the element
  - **ToArray()** - returns an array, containing all elements of the stack

Exampl


```
4
6
6
False
4
6
3
4
1
6
```

```
1  static void Main(string[] args)
2  {
3      var myStack = new Stack<int>();
4
5      myStack.Push(1);
6      myStack.Push(4);
7      myStack.Push(3);
8      myStack.Push(6);
9      myStack.Push(4);
10
11     Console.WriteLine(myStack.Pop());
12     Console.WriteLine(myStack.Peek());
13     Console.WriteLine(myStack.Peek());
14     Console.WriteLine(myStack.Contains(9));
15     Console.WriteLine(myStack.Count);
16     int [] myArray=myStack.ToArray();
17
18     foreach (int item in myStack)
19     {
20         Console.WriteLine(item);
21     }
22     Console.WriteLine(myStack.Pop());
23
24     Console.ReadKey();
25 }
```

Exercise 3:

- Write a program, which reads from the console N integers and prints them in reversed order. Use the Stack<T> class
  - Output

Enter 3 numbers:

45

67

89

The numbers in reverse order: 89 – 67 - 45

Exercise 4:

- Write a program that uses a stack to determine whether a string is a palindrome (i.e., the string is spelled identically backward and forward). The program should ignore spaces and punctuation.
  - Output

Enter the string: ABABA

The string is Palindrome

C# Queues

- A **queue** is a **First-In-First-Out (FIFO)** data structure.
  - The first element added to the queue will be the first one to be removed.
- Queues may be used to process messages as they appear or serve customers as they come. The first customer which comes should be served first.

- Elements added to the queue are **appended** at the **end of the queue**, and when elements are **extracted**, they are taken **from the beginning** of the queue (in the order they were added).
  - Thus the queue behaves like a list with two ends (head and tail), just like the queues for tickets.
- Like with the lists, the ADT queue could be implemented statically (as resizable array) and dynamically (as pointer-based linked list).

- Queue class provides the basic operations, specific for the data structure queue. Here are some of the most frequently used:
  - **Enqueue(T**) – inserts an element at the end of the queue
  - **Dequeue**() – retrieves the element from the beginning of the queue and removes it
  - **Peek**() – returns the element from the beginning of the queue without removing it
  - **Clear**() – removes all elements from the queue
  - **Contains(T**) – checks if the queue contains the element
  - **Count** – returns the amount of elements in the queue

Example


```
Message 1
Message 2
Message 2

Message 2
Message 3
Message 4
Message 5
```

```
static void Main(string[] args)
{
    var msgs = new Queue<string>();

    msgs.Enqueue("Message 1");
    msgs.Enqueue("Message 2");
    msgs.Enqueue("Message 3");
    msgs.Enqueue("Message 4");
    msgs.Enqueue("Message 5");

    Console.WriteLine(msgs.Dequeue());
    Console.WriteLine(msgs.Peek());
    Console.WriteLine(msgs.Peek());

    Console.WriteLine();

    foreach (string msg in msgs)
    {
        Console.WriteLine(msg);
    }
    Console.ReadKey();
}
```

- msgs.Enqueue("Message 1");
- msgs.Enqueue("Message 2");
- ...
  - The **Enqueue**() adds a message to the end of the queue.
- msgs.Dequeue();
  - The **Dequeue**() method removes and returns the item at the beginning of the queue.
- msgs.Peek();
  - The **Peek**() method returns the next item from the queue, but does not remove it from the collection.

Exercise 5:

- You are developing the help desk application of a company. Request of issues will be stored to be served on a first-come first-serviced basis. For each issue, the name of the employee and the description will be stored. Implement the queue on C# for issues.

C# List

- **List** is a strongly typed list of objects that can be accessed by index. It can be found under **System.Collections.Generic** namespace.
- List is the generic variant of ArrayList.
- When we create an object of type List, we indicate the type of the elements, which will be hold in the list,
  - i.e. we substitute the denoted by T type with some real data type (for example number or string)

- The following table lists the important properties and methods of List<T>

Property	Usage
Items	Gets or sets the element at the specified index
Count	Returns the total number of elements exists in the

Method	Usage
Add	Adds an element at the end of a List<T>.
Clear	Removes all the elements from a List<T>.
Find	Finds the first element based on the specified predicate function.
Insert	Inserts an element at the specified index in a List<T>.
Remove	Removes the first occurence of the specified element.
Remove	Removes the element at the specified index

```
5  using System.Collections.Generic;
6  namespace ConsoleApplication_Collection2
7  {
8      class Program
9      {
10         static void Main(string[] args)
11         {
12             var myList = new List<string>();
13             myList.Add("Java");
14             myList.Add("C#");
15             myList.Add("C");
16             myList.Add("C++");
17             myList.Add("Ruby");
18             Console.WriteLine(myList.Contains("C#"));
19             myList.Remove("C#");
20             Console.WriteLine(myList.Contains("C#"));
21             myList.Insert(3, "Python");
22             myList.Sort();
23             foreach (string lang in myList)
24             {
25                 Console.WriteLine(lang);
26             }
27             Console.ReadKey();
28         }
29     }
30 }
```

```
True
False
C
C++
Java
Python
Ruby
```

- **Line 5:** using System.Collections.Generic;
  - The List collection is located in the **System.Collections.Generic** namespace
- **Line 12:** var langs = new List<string>();
  - A generic dynamic array is created. We specify that we will work with strings with the type specified inside **<>** characters
- **Line 13-17**: myList.Add("Java");…
  - We add elements to the List using the **Add**() method

- **Line 18&20**:
  Console.WriteLine(langs.Contains("C#")); …
  - We check if the List contains a specific string using the **Contains**() method.
- **Line 19**: myList.Remove("C#");
  - We remove a strings from the List using the **Remove**() method
- **Line 21**: myList.Insert(3, “Python");
  - We insert a string at a specific location
- **Line 22**: myList.Sort();
  - We sort the elements using the **Sort**() method

C# SortedList

- **SortedList<T, T>** represents a collection of key/value pairs that are

```
static void Main(string[] args)
{
    var sorted = new SortedList<string, int>();
    sorted.Add("coins", 3);
    sorted.Add("books", 41);
    sorted.Add("spoons", 5);
    if (sorted.ContainsKey("books"))
    {
        Console.WriteLine("There are books in the list");
    }
    foreach (var pair in sorted)
    {
        Console.WriteLine(pair);
    }
    Console.ReadKey();
}
```

```
There are books in the list
[books, 41]
[coins, 3]
[spoons, 5]
```

- In SortedList, you can check whether the given pair is present or not using the following methods:
  - **Contains:** This method is used to check whether a SortedList object contains a specific key.
  - **ContainsKey:** This method is used to check whether a SortedList object contains a specific key.
  - **ContainsValue:** This method is used to check whether a SortedList object contains a specific value.

- The above code uses a sorted list to organize items.
- var sorted = new SortedList<string, int>();
  - The sorted list has string **keys** and integer **values**.
- if (sorted.ContainsKey("books"))
  - The sorted list has string keys and integer values.

C# LinkedList<T>

- **LinkedList <T>** class is a dynamic implementation of a generic **doubly linked list** built in .NET Framework.
  - Its elements contain a certain value and a pointer to the **previous** and the **next** element.
  - LinkedList only allows **sequential access**.
  - LinkedList allows for **constant-time** insertions or removals, but only sequential access of elements.
    - Because linked lists need extra storage for references, they are impractical for lists of small data items such as characters.

- When Should We Use LinkedList?
  - Using LinkedList is preferable when we have to **add** / **remove** elements at **both ends of the list** and when the access to the elements is consequential.
  - However, when we have to access the elements by index, then List is a more appropriate choice.
  - LinkedList has the same operations as in List, which makes the two classes interchangeable, but in fact List is used more often.

- **LinkedList** basic operations
  - The **append** operation is very fast, because the list always knows its last element (tail).
  - **Inserting** a new element at a random position in the list is very fast (unlike List) if we have a pointer to this position.
  - **Searching** for elements by index or by value in LinkedList is a slow operation, as we have to scan all elements consecutively by beginning from the start of the list.
  - **Removing** elements is a slow operation, because it includes searching.

- LinkedList provides 4 different methods to add nodes and these methods are:
  - **AddAfter:** This method is used to add a new node or value after an existing node in the LinkedList.
  - **AddBefore:** This method is used to add a new node or value before an existing node in the LinkedList.
  - **AddFirst:** This method is used to add a new node or value at the start of the LinkedList.
  - **AddLast:** This method is used to add a new node or value at the end of the LinkedList.

- LinkedList<T> class provides 5 different methods to remove elements and the methods are:
  - **Clear():** This method is used to remove all nodes from the LinkedList.
  - **Remove(LinkedListNode):** This method is used to remove the specified node from the LinkedList.
  - **Remove(T):** This method is used to remove the first occurrence of the specified value from the LinkedList.
  - **RemoveFirst():** This method is used to remove the node at the start of the LinkedList.
  - **RemoveLast():** This method is used to remove the node at the end of the LinkedList.

Examp


```
7
8
10
6
1
2
3
4
5
9
```

```
1  static void Main(string[] args)
2  {
3      var nums = new LinkedList<int>();
4
5      nums.AddLast(1);
6      nums.AddLast(2);
7      nums.AddLast(3);
8      nums.AddLast(4);
9      nums.AddLast(5);
10     nums.AddFirst(6);
11     nums.AddFirst(7);
12     nums.AddAfter(nums.First, 8);
13     nums.AddAfter(nums.Last, 9);
14     LinkedListNode<int> node = nums.Find(6);
15     nums.AddBefore(node, 10);
16
17     foreach (int num in nums)
18     {
19         Console.WriteLine(num);
20     }
21     Console.ReadKey();
22 }
```

- nums.AddLast(1); … nums.AddFirst(7);
  - We populate the linked list using the **AddLast**() and **AddFirst**() methods
- **nums.First** … **nums.Last**
  - We can find the first node using **First** and the last node using **Last** node
- LinkedListNode<int> node = nums.**Find**(6);
- nums.**AddBefore**(node, 10);
  - A LinkedList consists of nodes. We find a specific node and add an element before it.

Exercise 6:

- Write a program that reads a series of first names and stores them in a **LinkedList**. Do not store duplicate names. Allow the user to search for a first name.

C# Dictionary

- A **dictionary**, also called an associative array, is a collection of unique keys and a collection of values, where each key is associated with one value.
  - Retrieving and adding values is very fast. Dictionaries take more memory because for each value there is also a key.

Important Properties and Methods of Dictionary<T>

Property	Description
Count	Gets the total number of elements exists in the Dictionary<TKey,TValue>.
IsReadOnly	Returns a boolean indicating whether the Dictionary<TKey,TValue> is read-only.
Item	Gets or sets the element with the specified key in the Dictionary<TKey,TValue>.
Keys	Returns collection of keys of Dictionary<TKey,TValue>.
Values	Returns collection of values in Dictionary<TKey,TValue>.

Method	Description
Add	Adds an item to the Dictionary collection.
Add	Add key-value pairs in Dictionary<TKey, TValue> collection.
Remove	Removes the first occurrence of specified item from the Dictionary<TKey, TValue>.
Remove	Removes the element with the specified key.
ContainsKey	Checks whether the specified key exists in Dictionary<TKey, TValue>.
ContainsValue	Checks whether the specified key exists in Dictionary<TKey, TValue>.
Clear	Removes all the elements from Dictionary<TKey, TValue>.
TryGetValue	Returns true and assigns the value with specified key, if key does not exists then return false.

```
static void Main(string[] args)
{
    var domains = new Dictionary<string, string>();

    domains.Add("et", "Ethiopia");
    domains.Add("de", "Germany");
    domains.Add("ru", "Russia");
    domains.Add("us", "United States");

    Console.WriteLine(domains["ru"]);
    Console.WriteLine(domains["de"]);

    Console.WriteLine("Dictionary has {0} items",
        domains.Count);

    Console.WriteLine("Keys of the dictionary:");

    var keys = new List<string>(domains.Keys);

    foreach (string key in keys)
    {
        Console.WriteLine("{0}", key);
    }

    ...
```

```
        ...
        Console.WriteLine("Values of the dictionary:");

        var vals = new List<string>(domains.Values);
        foreach (string val in vals)
        {
            Console.WriteLine("{0}", val);
        }

        Console.WriteLine("Keys and values of the dictionary:");

        foreach (KeyValuePair<string, string> kvp in domains)
        {
            Console.WriteLine("Key = {0}, Value = {1}",
                kvp.Key, kvp.Value);
        }
        Console.ReadKey();
    }
```

- domains.Add(“et", “Ethiopia");
- domains.Add("de", "Germany");
- ...
  - We add some data to the dictionary.
    - The first string is the **key**.
    - The second is the **value**.
- Console.WriteLine(domains["ru"]);
- Console.WriteLine(domains["de"]);
  - Retrieve two values by their keys

- domains.Count;
  - We print the number of items by referring to the **Count** property.

C# collection initializers

- Collection initializers allow to specify elements to the collection during the object creation inside the **{}** brackets.

```
static void Main(string[] args)
{
    var vals = new List<int>() { 1, 2, 3, 4, 5, 6, 7 };
    int sum = vals.Sum();
    Console.WriteLine("The sum is: "+sum);
    Console.ReadKey();
}
```

```
The sum is: 28
```

Generic Classes and Methods

- ***Generic programming*** is a style of computer programming in which algorithms are written in terms of **to-be-specified-later types** that are then instantiated when needed for specific types provided as parameters.
- The **Advantages** and **Disadvantages** of Generics
  - Performance
  - Type safety
  - Binary code reuse
  - Code bloat
  - Naming guidelines

Naming Guidelines

- Here are naming guidelines for generic types:
  - Generic type names should be **prefixed** with the **letter T**.
  - If the generic type can be replaced by any class because there's no special requirement, and only one generic type is used, the character T is good as a generic type name:
    - **public class List <T>{ }**
    - **public class LinkedList <T>{ }**
  - If there's a special requirement for a generic type (for example, it must implement an interface or derive from a base class), or if two or more generic types are used, **descriptive names** should be used for the type names:
    - **public class SortedList <TKey, TValue>{ }**

Creating Generic Classes

- A generic class is defined similarly to a normal class with the generic type declaration.
  - The generic type can then be used within the class as a field member, or with parameter types of methods.

```
class MyClass<T>
    {
    //class definition
}
```

Example


```
1  class MyClass<T>
2  {
3      T myField;
4      public T MyField
5      {
6          get { return myField; }
7          set { myField = value; }
8      }
9      public MyClass(T myField)
10     {
11         this.myField = myField;
12     }
13
14     public T myMethod(T myParameter)
15     {
16         Console.WriteLine("My parameter is:" + myParameter);
17         return myParameter;
18     }
19 }
```

Generic Classes - Implementation


```
20  class Program
21  {
22      static void Main(string[] args)
23      {
24          MyClass<int> myObj1 = new MyClass<int>(123);
25          myObj1.MyField = 567;
26          myObj1.myMethod(myObj1.MyField);
27
28          MyClass<string> myObj2 = new MyClass<string>("C++ programming.");
29          myObj2.MyField = "C# programming.";
30          myObj2.myMethod(myObj2.MyField);
31
32          Console.ReadKey();
33      }
34  }
```

```
My parameter is:567
My parameter is:C# programming.
```

Generic Features

- When creating generic classes, you might need some additional C# keywords.
  - For example, it is not possible to assign null to a generic type. In this case, the keyword default can be used.
  - Default values
  - Constraints
  - Inheritance
  - Static members

Exercise 7: Write a generic Class?


```
1   public class Generator
2   {
3       // Display the value of an integer
4       public void Show(int value)
5       {
6           Console.WriteLine(value);
7       }
8       // Display the value of a double
9       public void Show(double value)
10      {
11          Console.WriteLine(value);
12      }
13      // Display the value of a character
14      public void Show(char value)
15      {
16          Console.WriteLine(value);
17      }
18  }
```

```
public class Exercise
{
    static int Main()
    {
        var exo = new Generator();
        //displays an integer
        exo.Show(246);
        // displays a character
        exo.Show('G');
        // displays a decimal
        exo.Show(355.65);
        // displays a string
        exo.Show("Hello"); // Error

        return 0;
    }
}
```

Questions?

Thank You