CSc 346
Object Oriented Programming

Ken Gamradt

Spring 2024

Chapter 8

Working with Common .NET Types

- This chapter is about some common types that are included with .NET
- These include types for:
  - manipulating numbers, text, and collections
  - improving working with spans, indexes, and ranges
  - optional online-only section
    - working with network resources

Working with numbers

Namespace	Example type(s)	Description
`System`	`SByte`, `Int16`, `Int32`, `Int64`, `Int128`	Integers; that is, zero, and positive and negative whole numbers.
`System`	`Byte`, `UInt16`, `UInt32`, `UInt64`, `UInt128`	Cardinals; that is, zero and positive whole numbers.
`System`	`Half`, `Single`, `Double`	Reals; that is, floating-point numbers.
`System`	`Decimal`	Accurate reals; that is, for use in science, engineering, or financial scenarios.
`System.Numerics`	`BigInteger`, `Complex`, `Quaternion`	Arbitrarily large integers, complex numbers, and quaternion numbers.

Working with text

Namespace	Type	Description
`System`	`Char`	Storage for a single text character
`System`	`String`	Storage for multiple text characters
`System.Text`	`StringBuilder`	Efficiently manipulates strings
`System.Text.RegularExpressions`	`Regex`	Efficiently pattern-matches strings

Joining, formatting, and other string members

Member	Description
`Trim`, `TrimStart`, `TrimEnd`	These methods trim whitespace characters such as space, tab, and carriage return from the start and/or end.
`ToUpper`, `ToLower`	These convert all the characters into uppercase or lowercase.
`Insert`, `Remove`	These methods insert or remove some text.
`Replace`	This replaces some text with other text.
`string.Empty`	This can be used instead of allocating memory each time you use a literal string value using an empty pair of double quotes ("").
`string.Concat`	This concatenates two string variables. The + operator does the equivalent when used between string operands.
`string.Join`	This concatenates one or more string variables with a character in between each one.
`string.IsNullOrEmpty`	This checks whether a string variable is null or empty.
`string.IsNullOrWhiteSpace`	This checks whether a string variable is null or whitespace; that is, a mix of any number of horizontal and vertical spacing characters, for example, tab, space, carriage return, line feed, and so on.
`string.Format`	An alternative method to string interpolation for outputting formatted string values, which uses positioned instead of named parameters.

Understanding the syntax of a regular expression

Symbol	Meaning	Symbol	Meaning
`^`	Start of input	`$`	End of input
`\d`	A single digit	`\D`	A single *non*-digit
`\s`	Whitespace	`\S`	*Non*-whitespace
`\w`	Word characters	`\W`	*Non*-word characters
`[A-Za-z0-9]`	Range(s) of characters	`\^`	^ (caret) character
`[aeiou]`	Set of characters	`[^aeiou]`	*Not* in a set of characters
`.`	Any single character	`\.`	. (dot) character

In addition, some common regular expression quantifiers that affect the previous symbols in a regular expression

Symbol	Meaning	Symbol	Meaning
`+`	One or more	`?`	One or none
`{3}`	Exactly three	`{3,5}`	Three to five
`{3,}`	At least three	`{,3}`	Up to three

Examples of regular expressions

Expression	Meaning
`\d`	A single digit somewhere in the input
`a`	The character "a" somewhere in the input
`Bob`	The word "Bob" somewhere in the input
`^Bob`	The word "Bob" at the start of the input
`Bob$`	The word "Bob" at the end of the input
`^\d{2}$`	Exactly two digits
`^[0-9]{2}$`	Exactly two digits
`^[A-Z]{4,}$`	At least four uppercase English letters in the ASCII character set only
`^[A-Za-z]{4,}$`	At least four upper or lowercase English letters in the ASCII character set only
`^[A-Z]{2}\d{3}$`	Two uppercase English letters in the ASCII character set and three digits only

`^[A-Za-z\u00c0-\u017e]+$`	At least one uppercase or lowercase English letter in the ASCII character set or European letters in the Unicode character set, as shown in the following list:
	`ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝ`
	`Þßàáâãäåæçèéêëìíîïðñòóôõö÷øùúûüýþÿıŒœŠšŸŽž`

`^d.g$`	The letter `d`, then any character, and then the letter `g`, so it would match both `dig` and `dog` or any single character between the `d` and `g`
`^d\.g$`	The letter `d`, then a dot `.`, and then the letter `g`, so it would match `d.g` only

Storing multiple objects in collections

Namespace	Example type(s)	Description
`System.Collections`	`IEnumerable`, `IEnumerable<T>`	Interfaces and base classes used by collections.
`System.Collections.Generic`	`List<T>`, `Dictionary<T>`, `Queue<T>`, `Stack<T>`	Introduced in C# 2.0 with .NET Framework 2.0. These collections allow you to specify the type you want to store using a generic type parameter (which is safer, faster, and more efficient).
`System.Collections.Concurrent`	`BlockingCollection`, `ConcurrentDictionary`, `ConcurrentQueue`	These collections are safe to use in multithreaded scenarios.
`System.Collections.Immutable`	`ImmutableArray`, `ImmutableDictionary`, `ImmutableList`, `ImmutableQueue`	Designed for scenarios where the contents of the original collection will never change, although they can create modified collections as a new instance.

Common features of all collections

- All collections implement the **ICollection** interface; this means that they must have a Count property to tell you how many objects are in them

```
namespace System.Collections;
0 references
public interface ICollection : IEnumerable {
    0 references
    int Count { get; }
    0 references
    bool IsSynchronized { get; }
    0 references
    object SyncRoot { get; }
    0 references
    void CopyTo(Array array, int index);
}
```

Common features of all collections

- All collections implement the **IEnumerable** interface, which means that they can be iterated using the foreach statement
- They must have a **GetEnumerator** method that returns an object that implements **IEnumerator**
- This means that the returned object must have **MoveNext** and **Reset** methods for navigating through the collection and a **Current** property containing the current item in the collection

Common features of all collections

```
namespace System.Collections;
0 references
public interface IEnumerable {
    0 references
    IEnumerator GetEnumerator();
}
1 reference
public interface IEnumerator {
    0 references
    object Current { get; }
    0 references
    bool MoveNext();
    0 references
    void Reset();
}
```

Common features of all collections

- As well as object-based collection interfaces, there are also generic interfaces and classes, where the generic type defines the type stored in the collection

```
namespace System.Collections.Generic;
0 references
public interface ICollection<T> : IEnumerable<T>, IEnumerable {
    0 references
    int Count { get; }
    0 references
    bool IsReadOnly { get; }
    0 references
    void Add(T item);
    0 references
    void Clear();
    0 references
    bool Contains(T item);
    0 references
    void CopyTo(T[] array, int index);
    0 references
    bool Remove(T item);
}
```

Working with lists

- Lists, that is, a type that implements IList<T>, are **ordered collections**

```
namespace System.Collections.Generic;
[DefaultMember("Item")] // aka "this" indexer
0 references
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable {
    T this[int index] { get; set; }
    0 references
    int IndexOf(T item);
    0 references
    void Insert(int index, T item);
    0 references
    void RemoveAt(int index);
}
```

Working with lists

- The [DefaultMember] attribute allows you to specify which member is accessed by default when no member name is specified
- To make IndexOf the default member, you would use [DefaultMember("IndexOf")]
- To specify the indexer, you use [DefaultMember("Item")]

Working with dictionaries

- Dictionaries are a good choice when each **value** (or object) has a unique subvalue that can be used as a **key** to quickly find a value in the collection later
- The key must be unique
- E.g., if you are storing a list of people, you could choose to use a government-issued identity number as the key
- Dictionaries are called **hashmaps** in other languages like Python and Java
- Think of the key as being like an index entry in a real-world dictionary
- It allows you to quickly find the definition of a word because the words are kept sorted
  - E.g., if we know we're looking for the definition of manatee, we will jump to the middle of the dictionary to start looking
    - because the letter m is in the middle of the alphabet
- Dictionaries in programming are similarly smart when looking something up
- They must implement the interface IDictionary<TKey, TValue>

Working with dictionaries

```
namespace System.Collections.Generic;
[DefaultMember("Item")] // aka "this" indexer
    : ICollection<KeyValuePair<TKey, TValue>>,
      IEnumerable<KeyValuePair<TKey, TValue>>, IEnumerable {
    TValue this[TKey key] { get; set; }
    0 references
    ICollection<TKey> Keys { get; }
    0 references
    ICollection<TValue> Values { get; }
    0 references
    void Add(TKey key, TValue value);
    0 references
    bool ContainsKey(TKey key);
    0 references
    bool Remove(TKey key);
    0 references
    bool TryGetValue(TKey key, [MaybeNullWhen(false)] out TValue value);
}
```

Dictionaries

- Items in a dictionary are instances of the struct, aka the value type KeyValuePair<TKey, TValue>, where TKey is the type of the key and TValue is the type of the value

```
namespace System.Collections.Generic;
1 reference
public readonly struct KeyValuePair<TKey, TValue> {
    0 references
    public KeyValuePair(TKey key, TValue value);
    0 references
    public TKey Key { get; }
    0 references
    public TValue Value { get; }
    [EditorBrowsable(EditorBrowsableState.Never)]
    0 references
    public void Deconstruct(out TKey key, out TValue value);
    0 references
    public override string ToString();
}
```

Sets, Stacks, and Queues

- **Sets** are a good choice when you want to perform set operations between two collections
  - Items in a set must be unique
- **Stacks** are a good choice when you want to implement **last-in, first-out** (**LIFO**) behavior
  - You can only directly access or remove the one item at the top of the stack, although you can enumerate to read through the whole stack of items
  - You cannot directly access the second item in a stack
- **Queues** are a good choice when you want to implement **first-in, first-out** (**FIFO**) behavior
  - You can only directly access or remove the item at the front of the queue, although you can enumerate to read through the whole queue of items
  - You cannot directly access the second item in a queue

Sets, Stacks, and Queues

Method	Description
`Add`	If the item does not already exist in the set, then it is added. Returns `true` if the item was added, and `false` if it was already in the set.
`ExceptWith`	Removes the items in the set passed as the parameter from the set.
`IntersectWith`	Removes the items not in the set passed as the parameter and in the set.
`IsProperSubsetOf`, `IsProperSupersetOf`, `IsSubsetOf`, `IsSupersetOf`	A subset is a set whose items are all in the other set. A proper subset is a set whose items are all in the other set but there is at least one item in the other set that is not in the set. A superset is a set that contains all the items in the other set. A proper superset is a set that contains all the items in the other set and at least one more not in the other set.
`Overlaps`	The set and the other set share at least one common item.
`SetEquals`	The set and the other set contain exactly the same items.
`SymmetricExceptWith`	Removes the items not in the set passed as the parameter from the set and adds any that are missing.
`UnionWith`	Adds any items in the set passed as the parameter to the set that are not already in the set.

Collection methods summary

- Each collection has a different set of methods for adding and removing items

Collection	Add methods	Remove methods	Description
List	`Add`, `Insert`	`Remove`, `RemoveAt`	Lists are ordered so items have an integer index position. `Add` will add a new item at the end of the list. `Insert` will add a new item at the index position specified.
Dictionary	`Add`	`Remove`	Dictionaries are not ordered so items do not have integer index positions. You can check if a key has been used by calling the `ContainsKey` method.
Stack	`Push`	`Pop`	Stacks always add a new item at the top of the stack using the `Push` method. The first item is at the bottom. Items are always removed from the top of the stack using the `Pop` method. Call the `Peek` method to see this value without removing it.
Queue	`Enqueue`	`Dequeue`	Queues always add a new item at the end of the queue using the `Enqueue` method. The first item is at the front of the queue. Items are always removed from the front of the queue using the `Dequeue` method. Call the `Peek` method to see this value without removing it.

Sorting collections

- There are multiple auto-sorting collections to choose from
- The differences between these sorted collections are often subtle but can have an impact on the memory requirements and performance of your application

Collection	Description
`SortedDictionary<TKey, TValue>`	This represents a collection of key/value pairs that are sorted by key. Internally it maintains a binary tree for items.
`SortedList<TKey, TValue>`	This represents a collection of key-value pairs that are sorted by key. The name is misleading because this is not a list. Compared to `SortedDictionary<TKey, TValue>`, retrieval performance is similar, it uses less memory, and insert and remove operations are slower for unsorted data. If it is populated from sorted data, then it is faster. Internally, it maintains a sorted array with a binary search to find elements.
`SortedSet<T>`	This represents a collection of unique objects that are maintained in a sorted order.

Working with network resources

Namespace	Example type(s)	Description
`System.Net`	`Dns`, `Uri`, `Cookie`, `WebClient`, `IPAddress`	These are for working with DNS servers, URIs, IP addresses, and so on.
`System.Net`	`FtpStatusCode`, `FtpWebRequest`, `FtpWebResponse`	These are for working with FTP servers.
`System.Net`	`HttpStatusCode`, `HttpWebRequest`, `HttpWebResponse`	These are for working with HTTP servers; that is, websites and services. Types from `System.Net.Http` are easier to use.
`System.Net.Http`	`HttpClient`, `HttpMethod`, `HttpRequestMessage`, `HttpResponseMessage`	These are for working with HTTP servers; that is, websites and services. You will learn how to use these in *Chapter 16, Building and Consuming Web Services*.
`System.Net.Mail`	`Attachment`, `MailAddress`, `MailMessage`, `SmtpClient`	These are for working with SMTP servers; that is, sending email messages.
`System.Net.NetworkInformation`	`IPStatus`, `NetworkChange`, `Ping`, `TcpStatistics`	These are for working with low-level network protocols.

Working with reflection and attributes

- **Reflection** is a programming feature that allows code to understand and manipulate itself
- An assembly is made up of up to four parts:
  - **Assembly metadata and manifest**: Name, assembly, and file version, referenced assemblies, …
  - **Type metadata**: Information about the types, their members, …
  - **IL code**: Implementation of methods, properties, constructors, …
  - **Embedded resources** (optional): Images, strings, JavaScript, …
- The metadata comprises items of information about your code
- The metadata is generated automatically from your code (e.g., information about the types and members) or applied to your code using attributes

Working with images

- ImageSharp is a third-party cross-platform 2D graphics library
- When .NET Core 1.0 was in development, there was negative feedback about the missing System.Drawing namespace for working with 2D images
- The **ImageSharp** project was started to fill that gap for modern .NET applications
  - In their official documentation for System.Drawing, Microsoft says, "The System.Drawing namespace is not recommended for new development due to not being supported within a Windows or ASP.NET service, and it is not cross-platform. ImageSharp and SkiaSharp are recommended as alternatives."
- ImageSharp also has NuGet packages for programmatically drawing images and working with images on the web, as shown in the following list:
  - SixLabors.ImageSharp.Drawing
  - SixLabors.ImageSharp.Web

Internationalizing your code

- Internationalization is the process of enabling your code to run correctly all over the world
- It has two parts: **globalization** and **localization**
- Globalization is about writing your code to accommodate multiple languages and region combinations
- The combination of a language and a region is known as a culture
- It is important for your code to know both the language and region
  - E.g., the date and currency formats are different in Quebec and Paris, despite them both using the French language

Internationalizing your code

- There are **International Organization for Standardization** (**ISO**) codes for all culture combinations
  - E.g., in the code da-DK, da indicates the Danish language and DK indicates the Denmark region
  - E.g., in the code fr-CA, fr indicates the French language and CA indicates the Canada region
- ISO is not an acronym
- ISO is a reference to the Greek word *isos* (which means equal)
- Localization is about customizing the user interface to support a language
- E.g., changing the label of a button to be Close (en) or Fermer (fr)
- Since localization is more about the language, it doesn't always need to know about the region, although ironically enough, standardization (en-US) and standardisation (en-GB) suggest otherwise