

SERIALIZATION AND COLLECTIONS

What would you learn/check about a
new class?

JSON to Object

Collections and Dictionaries

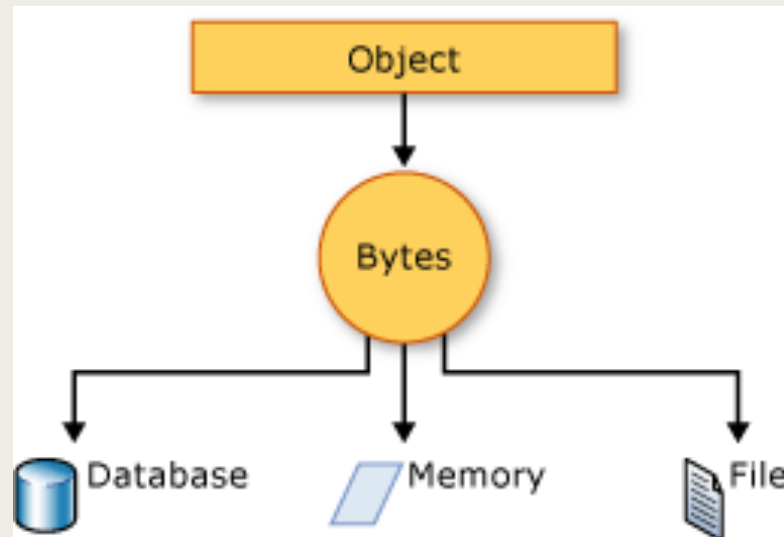
- What would you learn/check about a new class?

Flat Data Files

- A **flat file** database stores **data** in plain text format
 - *JSON*
 - *XML*
 - *CSV*

Serialization and Deserialization

- **Serialization** is the process of converting an **object** into a **stream of bytes** in order to store the object or transmit it to memory, a database, or a file.



JSON(JavaScript object notation)

```
{  
  'email' : 'new-email@hubspot.com',  
  'firstname': 'Mike',  
  'lastname': 'Newton'  
}
```

NuGet

- NuGet packages contain reusable code that other developers make available to you for use in your projects.
- A NuGet package is a single ZIP file with the .nupkg extension that contains **compiled code (DLLs)**, other files related to that code, and a descriptive manifest that includes information like the package's version number.

Assignment 4

Tasks:

- Read a JSON file to a string variable. Or directly assign a JSON string to the variable.
- [Deserialize](#) the JSON to be your class or a dynamic type
- Show the field names and values on the interface: the console or a form.

Assignment 4

Requirements:

- Design a few classes to do these steps, not only the application entrance class (For example: Program.cs or Form1.cs)

Create one class named as YourNameJson (For example: KevinLiJson). This class contains the method to deserialize the JSON string.

If you do not use dynamic, you need another class to be the destination class that is corresponding with the JSON data. For example: Student, with Name, Email etc. It could contain another member class, for example: DeskTop that then contains brand, year etc.

- Call the class and method(s) in the program.cs file or other files (for example: Form1.cs or FormJsonWork.cs), to do the deserialization and then show the detail values.

- Reading from a file or the JSON containing fields of an array or another class will get extra points.

- Showing the original JSON text will not get score points.

Assignment 4

Instruction

- Upload the class which name contains your name. Show me the application in the classroom.
- Use the Nuget of [Json.NET - Newtonsoft](#) or other classes to do the deserialization
- Sample codes:

```
Student s = new Student();
```

```
s = JsonConvert.DeserializeObject<Student> (sJson);
```

Or:

```
dynamic ss = JsonConvert.DeserializeObject(sJson);
```

Generics

- Generics are classes, structures, interfaces, and methods that have placeholders for one or more of the types that they store or use.

```
public class Generic<T>
{
    public T Field;
}
.
.
.
Generic <string> g = new Generic <string>();
g.Field = "A string";
```

- The .NET Framework includes several generic classes in the System.Collections.Generic namespace, including Dictionary, Queue, SortedDictionary, and SortedList.
- These classes work similarly to their nongeneric counterparts in System.Collections, but they offer improved performance and type safety.

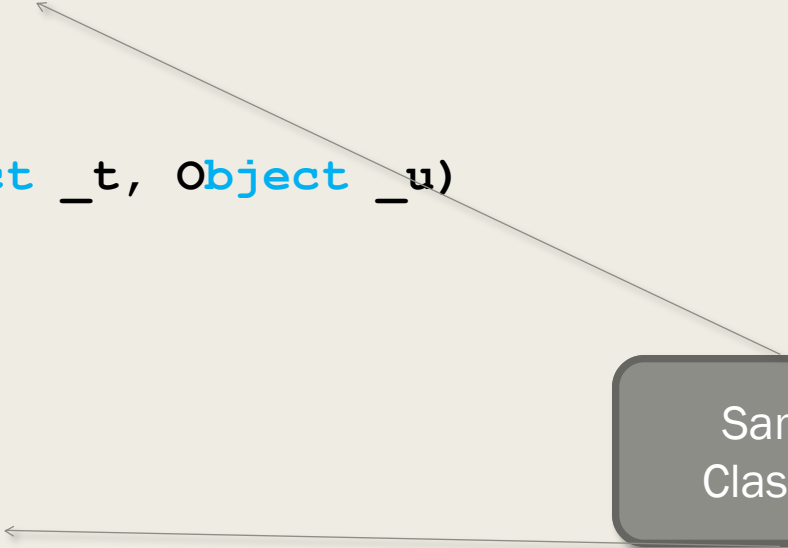
Generic Advantages

- **Reduced run-time errors** :The compiler cannot detect type errors when you cast to and from the Object class
- **Improved performance** : Casting requires boxing and unboxing, which requires processor time and slows performance. Using generics doesn't require casting or boxing

Creating Generic Types

```
class Obj
{
    public Object t;
    public Object u;
    public Obj(Object _t, Object _u)
    {
        t = _t;
        u = _u;
    }
}

class Gen<T, U>
{
    public T t;
    public U u;
    public Gen(T _t, U _u)
    {
        t = _t;
        u = _u;
    }
}
```



Same
Classes

Consuming Generic Types

- Specify the types for any generics used.

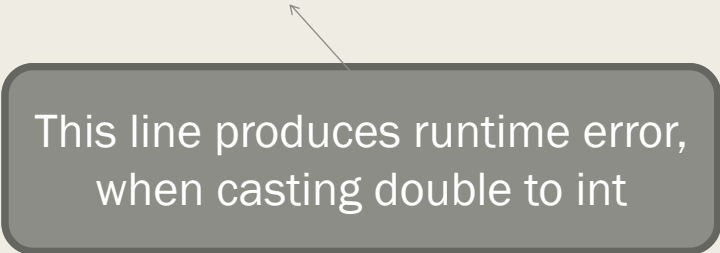
```
// Add two strings using the Obj class
Obj oa = new Obj ("Hello, ", "World!");
Console.WriteLine((string) oa.t + (string) oa.u);

// Add two strings using the Gen class
Gen<string, string> ga = new Gen<string, string>("Hello, ",
    , "World!");
Console.WriteLine(ga.t + ga.u);

// Add a double and an int using the Obj class
Obj ob = new Obj(10.125, 2005);
Console.WriteLine((double) ob.t + (int) ob.u);

// Add a double and an int using the Gen class
Gen<double, int> gb = new Gen<double, int>(10.125, 2005);
Console.WriteLine(gb.t + gb.u);
```

```
// Add a double and an int using the Obj  
class  
Obj oc = new Obj(10.125, 2005);  
Console.WriteLine((int)oc.t + (int)oc.u);
```



This line produces runtime error,
when casting double to int

Generic Constraints

- **Generics** would be limited if you could only write code that would compile for any class, because you would be limited to the capabilities of the base **Object** class.
- You can use constraints to overcome this limitation
- Constraints are applied using **where** clause

- Generics support four types of constraints:
 - 1) **Interface** : Allows only types that implement specific interfaces to be used as a generic type argument
 - 2) **Base class** : Allows only types that match or inherit from a specific base class to be used as a generic type argument
 - 3) **Constructor** : Requires types that are used as the type argument for your generic to implement a parameterless constructor
 - 4) **Reference or value type** : Requires types that are used as the type argument for your generic to be either a reference or a value type

```
class CompGen<T>  
    where T : IComparable
```

```
{
```

```
    public T t1;
```

```
    public T t2;
```

```
    public CompGen(T _t1, T _t2)
```

```
{
```

```
        t1 = _t1;
```

```
        t2 = _t2;
```

```
}
```

```
    public T Max()
```

```
{
```

```
        if (t2.CompareTo(t1) < 0)
```

```
            return t1;
```

```
        else
```

```
            return t2;
```

```
}
```

```
}
```



Constraint	Description
where T: struct	The type argument must be a value type. Any value type except Nullable can be specified.
where T : class	The type argument must be a reference type; this applies also to any class, interface, delegate, or array type.
where T : new()	The type argument must have a public parameterless constructor. When used together with other constraints, the new() constraint must be specified last.
where T : <u><base class name></u>	The type argument must be or derive from the specified base class.
where T : <u><interface name></u>	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic.
where T : U	The type argument supplied for T must be or <u>derive from the argument supplied for U.</u>

Collections

- A **collection** is any class that allows for gathering items into lists and for **iterating** through those items.
- Collections are data containers, which accept **variable quantities** of data
- Collections are stored in **System.Collections** and **System.Collections.Specialized** namespaces

Generic VS Nongeneric

- When working with objects of one specific type (or base type), use the **generic collections**
- The nongeneric collections can be found in [System.Collections](#), and generic collections can be found in [System.Collections.Generic](#).

.NET Collections

Class	Description
ArrayList	A simple collection that can store any type of object. ArrayList instances expand to any required capacity.
Queue	A first-in, first-out (FIFO) collection. You might use a Queue on a messaging server to store messages temporarily before processing or to track customer orders that need to be processed on a first-come, first-serve basis.
Stack	A last-in, first-out (LIFO) collection. You might use a Stack to track changes so that the most recent change can be undone.
StringCollection	Like ArrayList, except values are strongly typed as strings, and StringCollection does not support sorting.
BitArray	A collection of boolean values.

Performance concern: StringCollection vs List<String>

- `List<string>`: more idiomatic for most developers

ArrayList

- [ArrayList](#) is defined in [System.Collections](#)
- It Represents the **dynamically sized** array of objects
- It allows to add, remove, insert and sort items
- It contains items in order of addition
- It uses **index** identifier assigned to its objects to retrieve an element
- Enables programmers to retrieve items via index or by the order in which they are assigned

ArrayList example

```
ArrayList myAL = new ArrayList();  
myAL.Add("Hello");  
myAL.Add("World");  
myAL.Add("!");
```

```
foreach (string st in myList) {  
    Console.WriteLine("Value = {0}", st);  
}
```

Methods of ArrayList

- **Reverse** : method reverses sorting
- **BinarySearch** : array must be sorted to work
- **Contains** uses linear search
- **Sort** :
- **Add** : Adds an object to the end array
- **AddRange** : Adds the elements of an ICollection to the end of the array.
- Insert , Remove, RemoveAt, Clear

Custom IComparer

You can create your own custom **IComparer** implementations to control sort order

```
public class mySort : IComparer
{
    public int Compare(object x, object y)
    {
        string stx = (string)x;
        string sty = (string)y;

        return string.Compare(sty, stx);
    }
}

.
.
.
myAL.Sort(new mySort());
```

Binary Search

```
ArrayList myAL = new ArrayList();  
myAL.Add("The");  
myAL.Add("quick");  
myAL.Add("brown");  
myAL.Add("fox");
```

```
myAL.Sort();
```



Must be sorted first

```
Console.WriteLine(myAL.BinarySearch("fox"));
```

Queue Class

- Queue resides in `System.Collections`
- Queue implements `FIFO` strategy
- Uses `Enqueue/Dequeue` to add and remove elements
- Uses `Peek()` to peek at the first available element without removing it
- `Count` returns no. of elements;
- You can provide initial capacity and growth factor to constructor

Queue Example

```
Queue myQ = new Queue ();  
myQ.Enqueue ("Hello");  
myQ.Enqueue ("World");  
myQ.Enqueue ("!");  
myQ.Enqueue ("this");  
myQ.Enqueue ("is");  
myQ.Enqueue ("queue");  
Console.WriteLine (" (Dequeue) \t{0} \n",  
myQ.Dequeue ());
```

Stack Class

- Stack resides in `System.Collections`
- Stack implements `LIFO` strategy
- Uses `Push/Pop` to add and remove elements
- Uses `Peek()` to peek at the first available element without removing it

Stack Example

```
Stack myStack = new Stack ();  
myStack.Push("Hello");  
myStack.Push("World");  
myStack.Push("!");  
myStack.Push("this");  
myStack.Push("is");  
myStack.Push("stack");  
  
Console.WriteLine(myStack.Count);  
// Removes element from the Stack.  
Console.WriteLine("(pop) \t{0} \n", myStack.Pop());  
// Peek  
Console.WriteLine("(Peeking) \t{0} \n", myStack.Peek());
```


BitArray

- BitArray is located in System.Collections
- It represents array of bit values holding 1 or 0 values

BitArray Example

```
//Create BitArray
BitArray ba=new BitArray(3,true );
//Output count
Console.WriteLine(ba.Count );
foreach (object o in ba)
    Console.WriteLine(o);

ba[1]=false;
foreach (object o in ba)
    Console.WriteLine(o);
```

Why BitArray?

- A BitArray uses one bit for each value, while a bool[] uses one byte for each value.

Dictionaries

- Dictionary uses key/value pairs to store data
- Dictionary dynamically increase capacity
- Dictionary does not need integers for indexing
- There is no performance overhead for adding and removing elements
- The hash value of a key does not change during time and it can't be null.

Dictionary Classes

Dictionary Class	Description
HashTable	A dictionary of name-value pairs, which can be retrieved by name or index
SortedList	A dictionary of name-value pairs, which is sorted automatically <u>by key</u>
StringDictionary	A HashTable with name-value pairs implemented as strongly typed strings
ListDictionary	A dictionary optimized for small lists of objects with 10 items or less
HybridDictionary	A dictionary, which uses ListDictionary for storage when the number of items is small and automatically switches to HashTable when the list increases
NameValueCollection	A dictionary of name-value pairs of strings which allows retrieval by name or index

HashTable

- Each element is a key/value pair stored in a [DictionaryEntry](#) object.
- A key cannot be null, but a value can be
- A [HashTable](#) does not allow duplicate keys
(As Dictionary<TKey, TValue> implemented)
- Methods :
 - *Add, Remove, clone, contains, clear*
- Properties:
 - *Count, Keys, Values*

```
Hashtable openWith = new Hashtable();
```

```
openWith.Add("txt", "notepad.exe");
```

```
openWith.Add("bmp", "paint.exe");
```

```
openWith.Add("dib", "paint.exe");
```

```
openWith.Add("rtf", "wordpad.exe");
```

```
try
```

```
{
```

```
    openWith.Add("txt", "winword.exe");
```

```
}
```

```
catch
```

```
{
```

```
    Console.WriteLine("Key already exists.");
```

```
}
```

SortedList

- **SortedList** is a dictionary that consists of key/value pairs.
- **SortedList** can take an object of any type as its value but only strings as keys
- A **SortedList** element can be accessed by its key, or by its index.
- A **SortedList** object internally maintains two arrays to store the elements of the list, one for the keys and another for the values

- A **SortedList** does not allow duplicate keys
- The elements of a **SortedList** object are **sorted by the keys** either according to a specific **IComparer** implementation specified when the **SortedList** is created
- The index sequence is based on the sort sequence

```
// Creates and initializes a new SortedList.
SortedList mySL = new SortedList();
mySL.Add("Third", "!");
mySL.Add("Second", "World");
mySL.Add("First", "Hello");

Console.WriteLine("    Second:    {0}", mySL["Second"] );

for (int i = 0; i < mySL.Count; i++)
{
    Console.WriteLine("\t{0}:\t{1}", mySL.GetKey(i),
        mySL.GetByIndex(i));
}

foreach( DictionaryEntry de in mySL)
{
    Console.WriteLine("\t{0}:\t{1}", de.Key, de.Value);
}
```

ListDictionary

- The `ListDictionary` class (in the `System.Collections.Specialized` namespace) also provides similar functionality to `SortedList` but is optimized to perform best with lists of fewer than 10 items
- It is a simple implementation of `IDictionary` using a singly linked list
- Items in a `ListDictionary` are not in any guaranteed order; code should not depend on the current order

```
ListDictionary myCol = new ListDictionary();  
myCol.Add("Braeburn Apples", "1.49");  
myCol.Add("Fuji Apples", "1.29");  
myCol.Add("Gala Apples", "1.49");
```

HybridDictionary

- Implements **IDictionary** by using a **ListDictionary** while the collection is small, and then switching to a **HashTable** when the collection gets large.

StringDictionary

- The `StringDictionary` class (in the `System.Collections.Specialized` namespace) provides similar functionality to `SortedList`, **without the automatic sorting**, and requires both the keys and the values to be strings.
- The key is handled in a case-insensitive manner

NameValueCollection

- Represents a collection of associated `String` keys and `String` values that can be accessed either with the key or with the index.
- You can store multiple string values for a single key

NameValueCollection Example

Using System.Collections.Specialized

```
NameValueCollection nvc = new
```

```
NameValueCollection();
```

```
nvc.Add("Stack", "LIFO");
```

```
nvc.Add("Queue", "FIFO");
```

```
nvc.Add("Queue", "Line");
```

```
nvc.Add("Sorted List", "KVP");
```

```
foreach(string str in nvc.GetValues("Queue"))
```

```
{
```

```
    Console.WriteLine(str);
```

```
    // GetValues returns a collection
```

```
}
```


Generics

- Using **Generics**, you can create **strongly typed** collections for any class, including custom classes.
- **Generics** allow working with any type
- Using **Generics** enhances performance by reducing the potential for casting on insert and retrieval

Generic Types

Generic Class	NonGeneric Class	Description
List<T>	ArrayList StringCollection	A dynamically resizable list of items
Dictionary<T, <u>U</u> >	HashTable ListDictionary HybridDictionary OrderedDictionary NameValueCollection StringDictionary	A generic collection of name-value pairs
Queue<T>	Queue	A generic implementation of FIFO
Stack<T>	Stack	A Generic implementation of LIFO
<u>SortedList</u> <T, <u>U</u> >	SortedList	A generic implementation of sorted list or name-value pairs
Comparer<T>	Comparer	Comparer for generic values
LinkedList<T>	N/A	Generic doubly linked list
Collection<T>	CollectionBase	Base for generic collections
ReadOnlyCollection<T>	ReadOnlyCollectionBase	Base for generic read only set

List<T>

- List allow insertions and deletions at any location within the sequence.
- List<T> stores a group of items of type T.
- It enables duplicates and it quickly finds items.
- List has methods for adding and removing items, accessing items by index, and searching and sorting.
- List will increase the size of the array as necessary

```
public class List<T> : IList<T>, ICollection<T>, IList,
ICollection, IReadOnlyList<T> , IReadOnlyCollection<T> ,
IEnumerable<T>, IEnumerable
```

```
public interface IList<T> : ICollection<T>, IEnumerable<T>,
                        IEnumerable
```

```
{
    T this[int index] { get; set; }
    int IndexOf(T item);
    void Insert(int index, T item);
    void RemoveAt(int index);
}
```

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
    int Count { get; }
    bool IsReadOnly { get; }
    void Add(T item);
    void Clear();
    bool Contains(T item);
    void CopyTo(T[] array, int arrayIndex);
    bool Remove(T item);
}
```

```
List<string> listOfStrings =  
    new List<string> { "A", "B", "C", "D", "E"  
};
```

```
for (int x = 0; x < listOfStrings.Count; x++)  
{  
    Console.Write(listOfStrings[x]);  
}  
listOfStrings.Remove("A");
```

```
listOfStrings.Add("F");  
Console.WriteLine(listOfStrings.Count);
```

```
bool hasC = listOfStrings.Contains("C");
```

```
Console.WriteLine(hasC);
```

SortedList<T,U> Collection

- SortedList<TKey, TValue> is implemented as an array of key/value pairs, sorted by the key.
- Each element can be retrieved as a KeyValuePair<Tkey,Tvalue> object.

Generics with Custom Types

```
public class person
{
    string firstName;
    string lastName;
    public person( string _firstName, string _lastName)
    {
        firstName = _firstName;
        lastName = _lastName;
    }
    override public string ToString()
    {
        return firstName + " " + lastName;
    }
}

class Program
{
    static void Main(string[] args)
    {
        SortedList<string, person> sl = new SortedList<string, person>();
        sl.Add("One", new person("Mark", "Hanson"));
        sl.Add("Two", new person("Kim", "Akers"));
        sl.Add("Three", new person("Zsolt", "Ambrus"));
        foreach (person p in sl.Values)
            Console.WriteLine(p.ToString());
    }
}
```

Generic Queue<T> and Stack<T>

```
Queue < string > numbers = new Queue < string >();  
numbers.Enqueue("one");  
numbers.Enqueue("two");  
numbers.Enqueue("three");  
numbers.Enqueue("four");  
numbers.Enqueue("five");
```

```
// A queue can be enumerated without  
//disturbing its contents.  
foreach (string number in numbers)  
{  
    Console.WriteLine(number);  
}
```

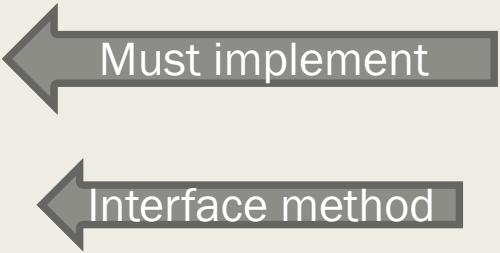

Generic List<T>

- The List<T> class is the generic equivalent of the [ArrayList](#) class.
 - *(Use List<T> whenever possible)*
- The List<T> class uses both an equality comparer and an ordering comparer.

- Methods such as `Contains`, `IndexOf`, `LastIndexOf`, and `Remove` use an equality comparer
- Methods such as `BinarySearch` and `Sort` use an ordering comparer for the list elements
- Calling `List.Sort` without any parameters requires the type to support the `IComparable` interface.

Generic List with Comparable

```
public class person : IComparable
{
    string firstName;
    string lastName;
    public int CompareTo(object obj)
    {
        person otherPerson = (person)obj;
        if (this.lastName != otherPerson.lastName)
            return this.lastName.CompareTo(otherPerson.lastName);
        else
            return this.firstName.CompareTo(otherPerson.firstName);
    }
    public person(string _firstName, string _lastName)
    {
        firstName = _firstName;
        lastName = _lastName;
    }
    override public string ToString()
    {
        return firstName + " " + lastName;
    }
}
```



Must implement

Interface method

Generic List with Comparable

```
class Program
{
    static void Main(string[] args)
    {
        List<person> l = new List<person>();
        l.Add(new person("Mark", "Hanson"));
        l.Add(new person("Kim", "Akers"));
        l.Add(new person("Michael", "Jones"));
        l.Add(new person("Zsolt", "Ambrus"));
        l.Sort();
        foreach (person p in l)
            Console.WriteLine(p.ToString());
        Console.ReadLine();
    }
}
```

HashSet<T>

- A **set** is a collection that contains no duplicate elements and has no order.
- Operations performed on a **set** are:
 - *checking if a set is a subset of another set,*
 - *selecting the elements that two sets have in common*
 - *selecting the elements that they don't have in common,*
 - *combining two sets*

```
public interface ISet<T> : ICollection<T>,
    IEnumerable<T>, IEnumerable
{
    bool Add(T item);
    void ExceptWith(IEnumerable<T> other);
    void IntersectWith(IEnumerable<T> other);
    bool IsProperSubsetOf(IEnumerable<T> other);
    bool IsProperSupersetOf(IEnumerable<T> other);
    bool IsSubsetOf(IEnumerable<T> other);
    bool IsSupersetOf(IEnumerable<T> other);
    bool Overlaps(IEnumerable<T> other);
    bool SetEquals(IEnumerable<T> other);
    void SymmetricExceptWith(IEnumerable<T> other);
    void UnionWith(IEnumerable<T> other);
}
```