Generic Collections in C#

Generic Collections

- A generic collection is strongly typed (type-safe).
- Meaning that you can only put one type of object into it.
- This eliminates type mismatches at runtime.

- Another benefit of type safety is that performance is better as there is no boxing or unboxing.
- There is no overhead of converting to and from type `object`.
- Generic collections belongs to `System.Collections.Generics` namespace.

- Few generic collection classes are:
  - List
  - Dictionary
  - Queue
  - Stack
  - Linked List
  - Sorted List
  - Sets

`List<T>` Generic Collection

- `List<T>` stores elements of the specified type and it grows automatically.
- `List<T>` can store multiple null and duplicate elements.
- `List<T>` can be accessed using indexer, for loop or foreach statement.
- LINQ can be used to query `List<T>` collection.
- `List<T>` is ideal for storing and retrieving large number of elements.

List<T> Generic Collection

- List<T> can be created and initialized in the following two ways.

```
List<int> myList = new List<int>();
myList.Add(10);
myList.Add(20);
myList.Add(30);
myList.Add(40);

List<int> myList = new List<int>() { 10, 20, 30, 40 };
```

List<T> Generic Collection

- Using the default constructor creates an empty list.
- As soon as elements are added to the list, the capacity of the list is extended to allow four elements.
- If the fifth element is added, the list is resized to allow eight elements.
- If eight elements are not enough, the list is resized again to contain 16 elements.
- With every resize, the capacity of the list is doubled.

```
List<int> myList = new List<int>();
```

List<T> Generic Collection

- If you know the number of elements in advance, that should be in the list; you can define the capacity with the constructor.
- The following example creates a collection with a capacity of 10 elements.
- If the capacity is not large enough for the elements added, the capacity is resized to 20 and then to 40 elements – doubled again.

```
List<int> myList = new List<int>(10);
```

List<T> Generic Collection

- You can get and set the capacity of a collection by using the `Capacity` property:

```
myList.Capacity = 20;
```

- The `Capacity` is not the same as the number of elements in the collection.
- The number of elements in the collection can be read with the `Count` property.

```
int numOfElements = myList.Count;
```

- If you are finished adding elements to the list and don't want to add any more, you can get rid of the unneeded capacity by invoking the `TrimExcess` method.
- However, because the relocation takes time, `TrimExcess` has no effect if the item count is more than 90 percent of capacity.

```
myList.TrimExcess();
```

`List<T>` Properties

Property	Usage
`Items`	Gets or sets the element at the specified index
`Count`	Returns the total number of elements in the `List<T>`

List<T> Methods

Method	Usage
Add	Adds an element at the end of a List<T>.
AddRange	Adds elements of the specified collection at the end of a List<T>.
BinarySearch	Search the element and returns an index of the element.
Clear	Removes all the elements from a List<T>.
Contains	Checks whether the specified element exists or not in a List<T>.
Find	Finds the first element based on the specified predicate function.
Foreach	Iterates through a List<T>.
Insert	Inserts an element at the specified index in a List<T>.
Remove	Removes the first occurrence of the specified element.
RemoveAt	Removes the element at the specified index.
RemoveRange	Removes all the elements that match with the supplied predicate function.
Sort	Sorts all the elements.
TrimExcess	Sets the capacity to the actual number of elements.

Add Elements into List

- Use the `Add()` method to add an element into a List collection.
- The following example adds `int` values into a `List<T>` of `int` type.

- `Add()` signature: `void Add(T item)`

```
List<int> myList = new List<int>();
myList.Add(10);
myList.Add(20);
myList.Add(30);
myList.Add(40);
```

Access List Collection

- Access individual items by using an indexer (i.e., passing an index in square brackets):

```
List<int> myList = new List<int>() { 10, 20, 30, 40 };

int item = myList[1];       // returns 20
```

- Use the `Count` property to get the total number of elements in the List.

```
List<int> myList = new List<int>() { 10, 20, 30, 40 };

Console.WriteLine("Total elements: " + myList.Count);
```

Loop Through List Collection

- Use a `foreach` loop:

```
List<int> myList = new List<int>() { 10, 20, 30, 40 };
foreach (var item in myList)
        Console.WriteLine(item);
```

- Use a `ForEach()` method:

```
List<int> myList = new List<int>() { 10, 20, 30, 40 };
myList.ForEach(item => Console.WriteLine(item));
```

- Use a `for` loop:

```
List<int> myList = new List<int>() { 10, 20, 30, 40 };
for (int i=0; i<myList.Count; i++)
        Console.WriteLine(myList[i]);
```

Insert into List

- The `Insert()` method inserts an element into a `List<T>` collection at the specified index.

- `Insert()` signature: `void Insert(int index, T item)`

```
List<int> myList = new List<int>() { 10, 20, 30, 40 };

myList.Insert(1, 11); // inserts 11 at index 1: after 10
```

Remove Elements from List

- The `Remove()` and `RemoveAt()` methods remove items from a `List<T>` collection.

- `Remove()` signature: `bool Remove(T item)`

- `RemoveAt()` signature: `void RemoveAt(int index)`

```
List<int> myList = new List<int>() { 10, 20, 30, 40 };

myList.Remove(10);  // removes 10 from a list

myList.RemoveAt(2); // removes the 3rd element (index starts from 0)
```

Searching

- There are different ways to search for elements in the collection:
  - You can get the index to the found item.
  - Or a boolean value if the `List<T>` contains the item.
  - Or the item itself.

- You can use methods such as:
  - `IndexOf()`
  - `Contains()`
  - `Exists()`
  - `Find()`

IndexOf()

- The method `IndexOf` requires an object as parameter and returns the index of the first occurrence of the item if it is found inside the collection.
- If the item is not found, -1 is returned.

```
List<int> myList = new List<int>() { 10, 20, 30, 40 };

int index = myList.IndexOf(30);  // returns 2
```

Contains()

- The method `Contains` requires an object as parameter and returns `true` if the item is found inside the collection.
- If the item is not found, `false` is returned.

```
List<int> myList = new List<int>() { 10, 20, 30, 40 };

bool isFound = myList.Contains(20); // returns true
```

Exists()

- The method `Exists` requires a predicate match or a lambda expression as parameter and returns `true` if the item is found inside the collection.
- If the item is not found, `false` is returned.

```
List<int> myList = new List<int>() { 10, 20, 30, 40 };

bool isFound = myList.Exists(num => num > 35); // returns true
```

Find()

- The method `Find` requires a predicate match or a lambda expression as parameter and returns the object itself if it is found inside the collection.
- If the object is not found, `null` is returned.

```
List<int> myList = new List<int>() { 10, 20, 30, 40 };

int num = myList.Find(num => num > 35); // returns 40
```

Dictionary<TKey, TValue> Generic Collection

- A Dictionary stores Key-Value pairs where the key must be unique.
- `TKey` denotes the type of key and `TValue` is the type of value.
- A C# Dictionary is similar to Java's `HashMap`.
- Before adding a `KeyValuePair` into a dictionary, check that the key does not exist using the `ContainsKey()` method.
- Use a `foreach` or `for` loop to iterate a dictionary.
- Use dictionary indexer to access individual item.

```
Dictionary<int, string> myDictionary = new Dictionary<int, string>();
```

Dictionary<TKey, TValue> Generic Collection

- Following are two examples of creating a dictionary.

```
Dictionary<int, string> myDictionary = new Dictionary<int, string>();

// Or

Dictionary<int, string> myDictionary = new Dictionary<int, string>()
                                       {
                                          {1, "One"},
                                          {2, "Two"},
                                          {3, "Three"}
                                       };
```

Dictionary<TKey, TValue> Properties

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>.

Dictionary<TKey, TValue> Methods

Method	Description
`void Add(TKey key, TValue value)`	Add key-value pairs in Dictionary<TKey, TValue> collection.
`void Remove(T item)`	Removes the first occurance of specified item from the Dictionary<TKey, TValue>.
`void Remove(TKey)`	Removes the element with the specified key.
`bool ContainsKey(TKey key)`	Checks whether the specified key exists in Dictionary<TKey, TValue>.
`bool ContainsValue(TValue value)`	Checks whether the specified key exists in Dictionary<TKey, TValue>.
`void Clear()`	Removes all the elements from Dictionary<TKey, TValue>.

Add Elements into Dictionary

- Use `Add()` method to add the key-value pair in dictionary.

- `Add()` Signature: `void Add(TKey, TValue)`

```
Dictionary<int, string> myDictionary = new Dictionary<int, string>();

myDictionary.Add(1, "One");
myDictionary.Add(2, "Two");
myDictionary.Add(3, "Three");
```

Access Dictionary Collection

- Dictionary can be used like an array to access its individual elements.
- Specify key (not index) to get a value from a dictionary using indexer like an array.

```
Console.WriteLine(myDictionary[1]);     // returns One
Console.WriteLine(myDictionary[2]);     // returns Two
```

- Indexer takes the key as a parameter.
- If the specified key does not exist, then a `KeyNotFoundException` will be thrown.
- It is a good idea to check if a key exists before retrieving the item.

```
if (myDictionary.ContainsKey(2))
{
        // do something
}
```

Loop Through Dictionary Collection

- Use `foreach` or `for` loop to iterate all the elements of dictionary.
- The dictionary stores key-value pairs.
- So you can use a `KeyValuePair<TKey, TValue>` type or an implicitly typed variable `var` in `foreach` loop.

```
foreach (KeyValuePair<int, string> item in myDictionary)
{
      Console.WriteLine("Key: " + item.Key);
      Console.WriteLine("Value: " + item.Value);
}
```

```
foreach (var item in myDictionary)
{
      Console.WriteLine("Key: " + item.Key);
      Console.WriteLine("Value: " + item.Value);
}
```

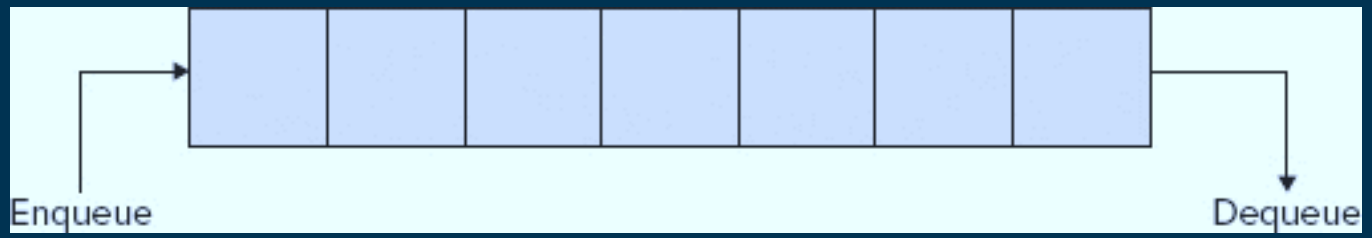
`Queue<T>` Generic Collection

- A queue is a collection whose elements are processed first-in-first-out (FIFO).
- Meaning the item that is put first in the queue is read first.
- Examples of queues are:
  - standing in line at the airport,
  - a human resources queue to process employee applicants,
  - print jobs waiting to be processed in a print queue.

- A queue is implemented with the `Queue<T>` class in the namespace `System.Collections.Generic`.

Queue<T> Generic Collection

- You cannot access the queue elements using an indexer.
- The queue just allows you to add an item to it, which is put at the end of the queue (with the `Enqueue` method), and to get items from the beginning of the queue (with the `Dequeue` method).



- The `Enqueue` method adds items to one end of the queue; the items are read and removed at the other end of the queue with the `Dequeue` method.
- Invoking the `Dequeue` method once more removes the next item from the queue.

Queue<T> Members

Members	Description
`Count`	Returns the number of items in the queue.
`Enqueue`	Adds an item to the end of the queue.
`Dequeue`	Reads and removes an item from the head of the queue. If there are no more items in the queue when the Dequeue method is invoked, an exception of type InvalidOperationException is thrown.
`Peek`	Reads an item from the head of the queue but does not remove the item.
`TrimExcess`	Resizes the capacity of the queue. The Dequeue method removes items from the queue, but it doesn't resize the capacity of the queue. To get rid of the empty items at the beginning of the queue, use the TrimExcess method.

Add Elements into Queue

- Use the `Enqueue()` method to add an element to the end of the queue.
- The following example adds `int` values into a `Queue<T>` of `int` type.

```
Queue<int> myQueue = new Queue<int>();

myQueue.Enqueue(10);
myQueue.Enqueue(20);
myQueue.Enqueue(30);
myQueue.Enqueue(40);
```

Remove Elements from Queue

- Use the `Dequeue()` method to remove an element from the head of the queue.

```
int num = myQueue.Dequeue();     // returns 10
```

- Use the `Peek()` method to read the element from the head of the queue, without removing it.

```
int num = myQueue.Peek();
```

Count Elements in a Queue

- Use the `Count` property to get the total number of elements in the queue.

```
int total = myQueue.Count;
```

Loop Through Queue

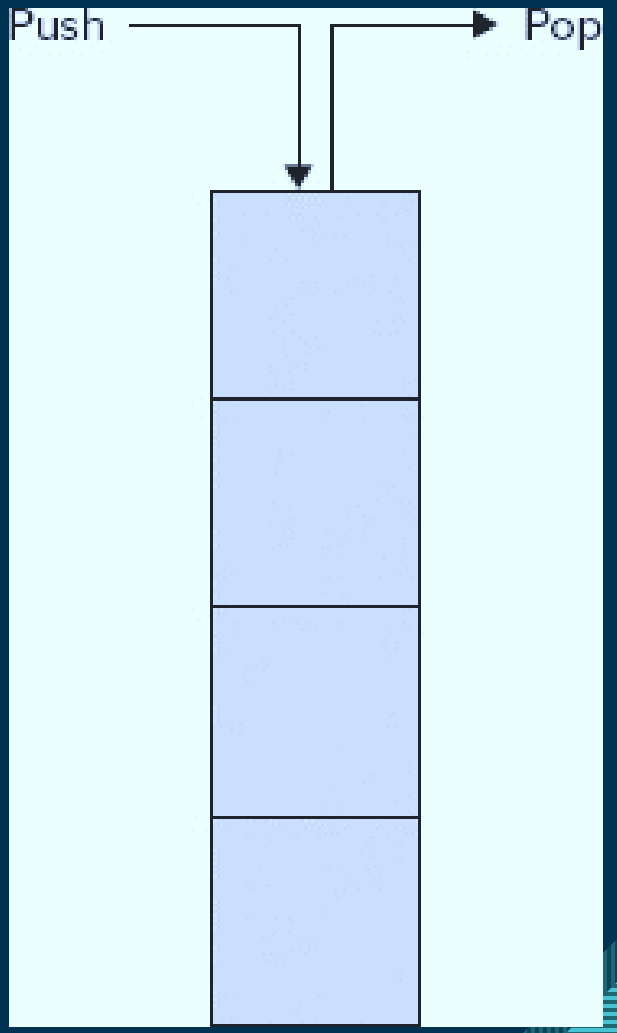
- Use a `foreach` loop:

```
foreach (var item in myQueue)
      Console.WriteLine(item);
```

Stack<T> Generic Collection

- A stack is somewhat similar to the queue.
- The difference lays in how the elements are added and removed.
- The item that is added last to the stack is read first, so the stack is a last-in-first-out (LIFO) container.

- The `Push` method adds an item to the stack, and the `Pop` method gets the item that was added last.



Stack<T> Members

Members	Description
`Count`	Returns the number of items in the stack.
`Push`	Adds an item on top of the stack.
`Pop`	Removes and returns an item from the top of the stack. If the stack is empty, an exception of type InvalidOperationException is thrown.
`Peek`	Returns an item from the top of the stack but does not remove the item.
`Contains`	Checks whether an item is in the stack and returns true if it is.

Add Elements into Stack

- Use the `Push()` method to add an element to the top of the stack.

```
Stack<int> myStack = new Stack<int>();

myStack.Push(10);
myStack.Push(20);
myStack.Push(30);
myStack.Push(40);
```

Remove Elements from Stack

- Use the `Pop()` method to remove an element from the top of the stack.

```
int num = myStack.Pop();    // returns 40
```

- Use the `Peek()` method to read the element from the top of the stack, without removing it.

```
int num = myStack.Peek();
```

Count Elements in a Stack

- Use the `Count` property to get the total number of elements in the stack.

```
int total = myStack.Count;
```

Loop Through Stack

- Use a `foreach` loop:

```
foreach (var item in myStack)
    Console.WriteLine(item);
```

Do It Yourself!

- A letter means enqueue and an asterisk means dequeue in the following sequence.
- Give the sequence of values returned by the dequeue operation when this sequence of operations is performed on an initially empty FIFO queue.

```
E A S * Y * Q U E * * * S T * * * I O * N * * *
```

Do It Yourself!

- **Exercise: Token System using Queue:**
- Write an app that implements a token system using a queue.
- As new customers come in, issue them a token number and add it to the queue.
- When a representative is free at the counter window, they can read the token number from the queue and handle that customer.

Do It Yourself!

- A letter means push and an asterisk means pop in the following sequence.
- Give the sequence of values returned by the pop operations when this sequence of operations is performed on an initially empty LIFO stack.

```
E A S * Y * Q U E * * * S T * * * I O * N * * *
```

Thank You

Copyright

The materials provided in class and in SLATE are protected by copyright. They are intended for the personal, educational uses of students in this course and should not be shared externally or on websites such as Chegg, Course Hero or OneClass. Unauthorized distribution may result in copyright infringement and violation of Sheridan policies.

References

Material has been taken from:

- Visual C# 2012: How to Program:
- https://learning.oreilly.com/library/view/visual-c-2012/9780133380170/ch20.html

- Professional C# 7 and .NET Core 2.0:
- https://learning.oreilly.com/library/view/professional-c-7/9781119449270/c05.xhtml