Generics

Generics

Generics allow the reuse of code across different types.
For example, let's declare a method that swaps the values of its two parameters:

```
static void Swap(ref int a, ref int b) {
  int temp = a;
  a = b;
  b = temp;
}
```

Our **Swap** method will work only for integer parameters. If we want to use it for other types, for example, doubles or strings, we have to overload it for all the types we want to use it with. Besides a lot of code repetition, it becomes harder to manage the code because changes in one method mean changes to all of the overloaded methods.
Generics provide a flexible mechanism to define a generic type.

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

In the code above, **T** is the name of our generic type. We can name it anything we want, but **T** is a commonly used name. Our Swap method now takes two parameters of type **T**. We also use the **T** type for our **temp** variable that is used to swap the values.

> Note the brackets in the syntax **<T>**, which are used to define a generic type.

Generic Methods

Now, we can use our **Swap** method with different types, as in:

```
static void Swap<T>(ref T a, ref T b) {
  T temp = a;
  a = b;
  b = temp;
}
static void Main(string[] args) {
  int a = 4, b = 9;
  Swap<int>(ref a, ref b);
  //Now b is 4, a is 9

  string x = "Hello";
  string y = "World";
  Swap<string>(ref x, ref y);
  //Now x is "World", y is "Hello"
}
```

Try It Yourself

When calling a generic method, we need to specify the type it will work with by using brackets. So, when **Swap<int>** is called, the **T** type is replaced by **int**. For **Swap<string>**, **T** is replaced by **string**. If you omit specifying the type when calling a generic method, the compiler will use the type based on the arguments passed to the method.

> Multiple generic parameters can be used with a single method.
> For example: **Func<T, U>** takes two different generic types.

Generic Classes

Generic types can also be used with classes.
The most common use for generic classes is with collections of items, where operations such as adding and removing items from the collection are performed in basically the same way regardless of the type of data being stored. One type of collection is called a stack. Items are "pushed", or added to the collection, and "popped", or removed from the collection. A stack is sometimes called a Last In First Out (LIFO) data structure.
For example:

```
class Stack<T> {
  int index=0;
  T[] innerArray = new T[100];
  public void Push(T item) {
    innerArray[index++] = item;
  }
  public T Pop() {
    return innerArray[--index];
  }
  public T Get(int k) { return innerArray[k]; }
}
```

The generic class stores elements in an array. As you can see, the generic type **T** is used as the type of the array, the parameter type for the **Push** method, and the return type for the **Pop** and **Get** methods.
Now we can create objects of our generic class:

```
Stack<int> intStack = new Stack<int>();
Stack<string> strStack = new Stack<string>();
Stack<Person> PersonStack = new Stack<Person>();
```

We can also use the generic class with custom types, such as the custom defined **Person** type.

> In a generic class we do not need to define the generic type for its methods, because the generic type is already defined on the class level.

Generic Classes

Generic class methods are called the same as for any other object:

```
Stack<int> intStack = new Stack<int>();
intStack.Push(3);
intStack.Push(6);
intStack.Push(7);

Console.WriteLine(intStack.Get(1));
//Outputs 6
```

Try It Yourself

Tap **Try It Yourself** to play around with the code!

C# Collections

A **collection** is used to group related objects. Unlike an array, it is **dynamic** and can also group objects. A collection can grow and shrink to accommodate any number of objects. Collection classes are organized into **namespaces** and contain built in methods for processing elements within the collection.

A collection **organizes** related data in a computer so that it can be used efficiently.
Different kinds of collections are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, **Dictionaries** are used to represent connections on social websites (such as Twitter, Facebook), queues can be used to create task schedulers, **HashSets** are used in searching algorithms, etc.

A collection typically includes methods to **add**, **remove**, and **count** objects. The **for** statement and the **foreach** statement are used to **iterate** through collections. Since a collection is a class you must first declare an instance of the class before you can add elements to that collection. For example:

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

Collections provide a more flexible way to work with groups of objects. Unlike arrays, the group of objects you work with can grow and shrink dynamically as the needs of the application change.

Generic Collections

Generic collections are the preferred type to use as long as every element in the collection is of the same data type. Only desired data types can be added to a generic collection and this is enforced by using strong typing which reduces the possibility of errors.
The .NET Framework provides a number of generic collection classes, useful for storing and manipulating data.
The **System.Collections.Generic** namespace includes the following generic collections:
- **List<T>**
- **Dictionary<TKey, TValue>**
- **SortedList<TKey, TValue>**
- **Stack<T>**
- **Queue<T>**
- **Hashset<T>**

To access a generic collection in your code, you will need to include the statement: **using Systems.Collections.Generic;**

Non-Generic Collections

Non-generic collections can store items that are of type Object. Since an Object data type can refer to any data type, you run the risk of unexpected outcomes. Non-generic collections may also be slower to access as well as execute.
The **System.Collections** namespace includes the following non-generic collections:
- **ArrayList**
- **SortedList**
- **Stack**
- **Queue**
- **Hashtable**
- **BitArray**

Because non-generic collections are error prone and less performant, it is recommended to always use generic collections from the System.Collections.Generic namespace if available and to avoid using legacy collections from the System.Collections namespace.

List<T>

A **list** is similar to an array, but the elements in a list can be inserted and removed **dynamically**. The C# generic collection **List<T>** class requires all elements be of the same type **T**.

List<T> properties and methods include:
Count A property that gets the number of elements contained in the list.
Item[int i] Gets or sets the element in the list at the index i. Item is the indexer and is not required when accessing an element. You only need to use the brackets [] and the index value inside the brackets.
Add(T t) Adds an element **t** to the end of the list.
RemoveAt(int index) Removes the element at the specified position (index) from the list.
Sort() Sorts elements in the list.

Now let's try **List<T>**:

```
List<int> li = new List<int>();
li.Add(59);
...
li.RemoveAt(1);

Console.Write("\nList: ");
for (int x = 0; x < li.Count; x++)
  Console.Write(li[x] + " ");
li.Sort();
Console.Write("\nSorted: ");
for (int x = 0; x < li.Count; x++)
  Console.Write(li[x] + " ");
```

Try It Yourself

Additional List<T> properties and methods are listed below. Try them out by adding them to the List<T> example code above.

Capacity - A property that gets the number of elements the list can hold before needing to be resized.

Clear() - Removes all the elements from the list.

TrimExcess() - Sets the capacity to the actual number of elements in the list. This is useful when trying to reduce memory overhead.

AddRange(IEnumerable coll) - Adds the elements of collection coll with elements of the same type as List<T> to the end of the list. IEnumerable is the collections interface that supports simple iteration over the collection.

Insert(int i, T t) - Inserts an element t at a specific index i in the list.

InsertRange(int i, IEnumerable coll) - Inserts the elements of a collection coll at a specified index i in the list. IEnumerable is the collections interface that supports simple iteration over the collection.

Remove(T t) - Removes the first occurrence of the object t from the list.

RemoveRange(int i, int count) - Removes a specified number of elements count from the list starting at a specified index i.

Contains(T t) - Returns true if the specified element t is present in the list.

IndexOf(T t) - Returns the index of the first occurrence of the element t in the list.

Reverse() - Reverses the order of the elements in the list.

ToArray() - Copies the elements of the list into a new array.

Tap **Try It Yourself** and modify the code to insert elements using the for loop. Remember, you need to include the statement: **using Systems.Collections.Generic;** to use List<T>.

SortedList<K, V>

A **sorted list** is a collection of **key/value pairs** that are sorted by key. A key can be used to access its corresponding value in the sorted list.

The C# generic collection **SortedList<K, V>** class requires all element key/value pairs to be of the same type **K, V**. Duplicate keys are **not permitted**, which ensures that every key/value pair is unique.

SortedList<K, V> properties include:
Count - Gets the number of key/value pairs contained in the sorted list.
Item[K key] - Gets or sets the value associated the specified key contained in the sorted list. Item is the indexer and is not required when accessing an element. You only need to use the brackets [] and the key, value.
Keys - Gets a sorted and indexed collection containing only the keys in the sorted list.

SortedList<K, V> methods include:
Add(K key, V value) - Adds an element with a specific key, value pair into the sorted list.
Remove(K key) - Removes the element with the specific key, value pair associated with the specified key from the sorted list.

Now let's try **SortedList<K, V>**:

```
SortedList<string, int> sl = new SortedList<string, int>();
sl.Add("Solo", 59);
sl.Add("A", 95);
sl.Add("Learn", 72);
sl.Remove("A");
Console.WriteLine("Sorted List: ");
foreach (string s in sl.Keys)
  Console.WriteLine(s + ": " + sl[s]);  // Learn: 72  Solo: 59
Console.WriteLine("\nCount: " + sl.Count);  // 2
```

Try It Yourself

Here are additional **SortedList<K, V>** properties and methods:
Values - Gets a sorted and indexed collection of the values in the sorted list.
Clear() - Removes all the elements from the sorted list.
ContainsKey(K key) - Returns true when the specified key is present in the sorted list.
ContainsValue(V value) - Returns true when a specified value is present in the sorted list.
IndexOfKey(K key) - Returns the index of the specified key within the sorted list.
IndexOfValue(V value) - Returns the index of the specified value within the sorted list.

Tap **Try It Yourself** and modify the code to remove an element from the SortedList that doesn't belong to the list.

BitArray

A bit array is a **collection of bits**. The value of a bit can be either **0** (off/*false*) or **1** (on/*true*).
Bit arrays compactly store bits. Most commonly, they are used to represent a simple group of boolean flags or an ordered sequence of boolean values.

BitArray properties include:
Count - Gets the number of bits in the bit array.
IsReadOnly - Gets a value indicating if the bit array is read only or not.

BitArray methods include:
Get(int i) - Gets the value of the bit at a specified position i in the bit array.
Set(int i, bool value) - Sets the bit at a specified position i to a specified value in the bit array.
SetAll(bool value) - Sets all the bits to a specified value in the bit array.
And(BitArray ba) - Performs the bitwise AND operation on the elements of the bit array object

with a specified bit array ba.
Or(BitArray ba) - Performs the bitwise OR operation on the elements of the bit array and the specified bit array ba.
Not() - Inverts the bit values of the bit array.
Xor(BitArray ba) - Performs the bitwise XOR operation on the elements of the current bit array object and the elements in the specified bit array ba.

This example demonstrates some properties and methods of the **BitArray** class:

```
static void Main(string[] args) {
  BitArray ba1 = new BitArray(4);
  BitArray ba2 = new BitArray(4);

  ba1.SetAll(true);
  ba2.SetAll(false);

  ba1.Set(2, false);
  ba2.Set(3, true);

  PrintBarr("ba1", ba1);
  PrintBarr("ba2", ba2);
  Console.WriteLine();

  PrintBarr("ba1 AND ba2", ba1.And(ba2));
  PrintBarr("    NOT ba2", ba2.Not());
}

static void PrintBarr(string name, BitArray ba) {
  Console.Write(name + " : ");
  for (int x = 0; x < ba.Length; x++)
    Console.Write(ba.Get(x) + " ");
  Console.WriteLine();
}
```

Try It Yourself

For example, BitArrays can be used in image processing to store the individual bits of a gray-scale image.

Tap **Try It Yourself** and modify the code to apply the AND operation to five bit arrays.

Stack<T>

A **stack** is a **Last In, First Out** (**LIFO**) collection of elements where the last element that goes into the stack will be the first element that comes out.

Inserting an element onto a stack is called **pushing**. Deleting an element from a stack is called **popping**. Pushing and popping can be performed only at the **top** of the stack.

Stacks can be used to create undo-redo functionalities, parsing expressions (infix to postfix/prefix conversion), and much more.

The C# generic collection **Stack<T>** class requires all elements to be of the same type **T**.

Stack<T> properties include:
Count - Returns the number of elements in the stack.

Stack<T> methods include:
Peek() - Returns the element at the top of the stack without removing it.
Pop() - Returns the element at the top of the stack and removes it from the stack.
Push(T t) - Inserts an element t at the top of the stack.

Now let's try **Stack<T>**:

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

s.Push(59);
...
Console.Write("Stack: ");
foreach (int i in s)
  Console.Write(i + " ");  // 65  72  59
Console.Write("\nCount: " + s.Count);  // 3

Console.Write("\nTop: " + s.Peek());  // 65
Console.Write("\nPop: " + s.Pop());  // 65

Console.Write("\nStack: ");
foreach (int i in s)
  Console.Write(i + " ");  // 72  59
Console.Write("\nCount: " + s.Count);  // 2
```

Try It Yourself

Here are additional **Stack<T>** methods:
Clear() - Removes all the elements from the stack.
Contains(T t) - Returns true when the element **t** is present in the stack.
ToArray() - Copies the stack into a new array.

> Tap **Try It Yourself** and modify the code to pop elements from an empty Stack.

Queue<T>

A **queue** is a **First In, First Out** (**FIFO**) collection of elements where the first element that goes into a queue is also the first element that comes out.

Inserting an element into a queue is referred to as **Enqueue**. Deleting an element from a queue is referred to as **Dequeue**.

> Queues are used whenever we need to manage objects in order starting with the first one in.
> Scenarios include printing documents on a printer, call center systems answering people on hold people, and so on.

The C# generic collection **Queue<T>** class requires that all elements be of the same type **T**.

Queue<T> properties include:
Count - Gets the number of elements in the queue.

And methods include:
Dequeue() - Returns the object at the beginning of the queue and also removes it.
Enqueue(T t) - Adds the object t to the end of the queue.

Now let's try **Queue<T>**:

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

q.Enqueue(5);
q.Enqueue(10);
q.Enqueue(15);
Console.Write("Queue: ");
foreach (int i in q)
  Console.Write(i + " ");  // 5  10  15
Console.Write("\nCount: " + q.Count);  // 3

Console.Write("\nDequeue: " + q.Dequeue()); // 5

Console.Write("\nQueue: ");
foreach (int i in q)
  Console.Write(i + " ");  // 10  15
Console.Write("\nCount: " + q.Count);  // 2
```

Try It Yourself

Here are additional **Queue<T>** methods:
Clear() - Removes all objects from the queue.
Contains(T t) - Returns true when the element t is present in the queue.
Peek() - Returns the object at the beginning of the queue without removing it.
ToArray() - Copies the queue into a new array.

Tap **Try It Yourself** and modify the code to insert the Queue elements from a Stack.

Dictionary<U, V>

A **dictionary** is a collection of unique key/value pairs where a key is used to access the corresponding value. Dictionaries are used in database indexing, cache implementations, and so on.

The C# generic collection **Dictionary<K, V>** class requires all key/value pairs be of the same type **K, V**. Duplicate keys are **not permitted** to ensure that every key/value pair is unique.

Dictionary<K, V> properties include:
Count - Gets the number of key/value pairs contained in the dictionary.
Item[K key] - Gets the value associated with the specified key in the dictionary. Item is the indexer and is not required when accessing an element. You only need to use the brackets [] and key value.
Keys - Gets an indexed collection containing only the keys contained in the dictionary.

Dictionary<K, V> methods include:
Add(K key, V value) - Adds the key, value pair to the dictionary.
Remove(K key) - Removes the key/value pair related to the specified key from the dictionary.

Now let's try **Dictionary<K, V>**:

```
Dictionary<string, int> d = new Dictionary<string, int>();
d.Add("Uno", 1);
d.Add("One", 1);
...
d.Remove("One");  // Remove key-value pair One, 1
...
Console.WriteLine("Dictionary: ");
foreach (string s in d.Keys)
   Console.WriteLine(s + ": " + d[s]);  // Uno: 1  Deux: 2
Console.WriteLine("\nCount: {0}", d.Count); // 2
```

Try It Yourself

In the above example, the dictionary **d** uses strings as it's keys and integers as the values.

Here are the additional **Dictionary<K, V>** properties and methods:
Values - Gets an indexed collection containing only the values in the dictionary.
Clear() - Removes all the key/value pairs from the dictionary.
ContainsKey(K key) - Returns true if the specified key is present in the dictionary.
ContainsValue(V value) - Returns true if the specified value is present in the dictionary.

> Tap **Try It Yourself** and modify the code to add duplicate values to the Dictionary.

HashSet<T>

A **hash set** is a set of unique values where duplicates are not allowed.

C# includes the **HashSet<T>** class in the generic collections namespace. All **HashSet<T>** elements are required to be of the same type **T**.
Hash sets are different from other collections because they are simply a set of values. They do not have index positions and elements cannot be ordered.

> The HashSet<T> class provides high-performance set operations. HashSets allow fast lookup, addition, and removal of items, and can be used to implement either dynamic sets of items or lookup tables that allow finding an item by its key (e.g., finding the phone number of a person by the last name).

HashSet<T> **properties** include:
Count Returns the number of values in the hash set.

And **methods** include:
Add(T t) Adds a value (t) to the hash set.
IsSubsetOf(ICollection c) Returns true if the hash set is a subset of the specified collection (c).

Now let's try **HashSet<T>**:

```
HashSet<int> hs = new HashSet<int>();

hs.Add(5);
...
Console.Write("\nHashSet: ");
foreach (int i in hs)
  Console.Write(i + " "); // 5  10  15  20  *elements may be in any order
Console.Write("\nCount: " + hs.Count);  // 4

HashSet<int> hs2 = new HashSet<int>();
hs2.Add(15);
hs2.Add(20);
Console.Write("\n{15, 20} is a subset of {5, 10, 15, 20}: " + hs2.IsSubsetOf(hs)); // True
```

Try It Yourself

Here are additional HashSet<T> methods:
Remove(T t) Removes the value (t) from the hash set.
Clear() Removes all the elements form the hash set.
Contains(T t) Returns true when a value (t) is present in the hash set.
ToString() Creates a string from the hash set.
IsSupersetOf(ICollection c) Returns true if the hash set is a superset of the specified collection.
UnionWith(ICollection c) Applies set union operation on the hash set and the specified collection (c).
IntersectWith(ICollection c) Applies set intersection operation on the hash set and the specified collection (c).
ExceptWith(ICollection c) Applies set difference operation on the hash set and the specified collection (c).

> Tap **Try It Yourself** and modify the code to print the HashSet values in increasing order.

End.