Generic Classes & Methods

- ***Generics*** allow the programmer to create classes that contain properties or methods that can work with any data type or fields that can be any data type. This allows for greater code reuse.
- The typed collections – like List<T> and Dictionary<TKey, TValue> – are examples of generic classes. The List<T> can hold numbers or strings – the type (T) is set when you create the list collection.
- The programmer can create generic classes, structures, interfaces, and methods (among other C# constructs).

Defining a Generic Class

- Creating generic classes is not something that is done often. The most common use is to add functionality to Microsoft's generic collections.
- To define a generic class, you add angle brackets after the name of the class. Inside the angle brackets, you specify a name that will represent the data type when the class is instantiated – this is known as the *type parameter* of the class.
- When a generic class accepts one type parameter, most programmers use the letter **T** as the name for the type parameter.
- When a generic class accepts more than one type parameter, the convention is to start with the letter **T** followed by a meaningful name – like **TKey** and **TValue**.

General Form of a Generic Class

```
public class ClassName1<T>
{
    private T fieldName;
    public T PropertyName
    {
        // get and/or set accessors
    }
    public T MethodName(T paramName)
    {
        // return value of type T
    }
}
```

```
public class ClassName2<Type1, Type2>
{
    private Type1 field1;
    private Type2 field2;
    public Type1 Property1
    {
        // get and/or set accessors
    }
    public Type2 Property2
    {
        // get and/or set accessors
    }
    public void Method(Type1 var1, Type2 var2)
    {//code}
}
```

Instantiating a Generic Class

```
// Can instantiate the generic class with any data type – value or reference type
ClassName1<int> object1 = new ClassName1<int>();
ClassName1<PowerBoat> object2 = new ClassName1<PowerBoat>();
ClassName2<string, double> object3 = new ClassName2<string, double>();
ClassName2<PowerBoat, SailBoat> object4 = new ClassName2<PowerBoat, SailBoat>();
```

Constraints

- To restrict the type of data that a generic class can accept, the programming can specify a constraint for the type parameter.
- By constraining the type parameter, you increase the number of allowable operations and method calls to those supported by the constraining type and all types in its inheritance hierarchy.
- Constraints are specified by using the *where* contextual keyword.

The following table lists the five most common types of constraints:

Constraint	Description
where T: struct	The type argument must be a value type.
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 : <base class name>	The type argument must be or derive from the specified base class.
where T : <interface name>	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic.

Examples: Generic Class Headers with Constraints

```
// a generic class that can accept only value types – like int, double, struct, enum, etc.
public class Boat<T> where T: struct
// a generic class that can accept only Account objects or objects of classes derived from Account
public class BankAccount<T> where T: Account
// Multiple constraints can be applied to the same type parameter, and the constraints themselves can be generic types, as follows:
public class EmployeeList<T> where T : Employee, IEmployee, IComparable<T>, new()
```

Generic Methods

- Generic methods can be created in either generic or non-generic classes.
- A generic method is a method that is declared with type parameters, as follows:

```
public void MethodName<T>(T param1, T param2)
```

- Within a generic class, non-generic methods can access the class-level type parameters, as follows:

```
public class Class1<T>
{
    public void Method1(T param1, T param2){// code}
}
```

- If you define a generic method that takes the same type parameters as the containing class, the compiler generates a warning because within the method scope, the argument supplied for the inner T hides the argument supplied for the outer T. If you require the flexibility of calling a generic class method with type arguments other than the ones provided when the class was instantiated, consider providing another identifier for the type parameter of the method, as follows:

```
public class Class1<T>
{
            // Warning: Method type parameter overrides Class type parameter
    public void Method1<T>(T param1) { }
}
```

```
public class Class2<T>
{
    //No warning
    public void Method2<U>(U param1) { }
}
```

- Generic methods can also list constraints like generic classes:

```
public void MethodName<T>(T param1, T param2) where T: struct
```

  - Generic methods can be very limiting since the value type **T** is very general – C# will not allow you to compare or add data together.
  - Using constraints allows greater options in how the data can be used – but choosing the proper constraints is important. Some of the more valuable constraints are interfaces – specifying a base class constraint can also increase your options.

Example: Generic Method in a Form Class

```
// a generic method that will accept any data type reference
public void Swap<T>(ref T left, ref T right)
{
    // swap one value for the other
    T temp = left;
    left = right;
    right = temp;
}

// code that uses the generic method
string display = "";
int a = 3, b = 4;
display += "Before Swap...A = " + a + ", B = " + b + "\n";
Swap<int>(ref a, ref b);
display += "After Swap...A = " + a + ", B = " + b + "\n\n";

string first = "Peter", last = "Vanscoy";
display += "Before Swap...First = " + first + ", Last = " + last + "\n";
Swap<string>(ref first, ref last);
display += "After Swap...First = " + first + ", Last = " + last;

txtTextBox.Text = display;
```

Output:

```
Before Swap…A = 3, B = 4
After Swap…A = 4, B = 3

Before Swap…First = Peter, Last = Vanscoy
After Swap…First = Vanscoy, Last = Peter
```