Topic: Generics

Presenter: Paul Stauskas

Generics are one of the most important language concepts to understand. Without a thorough understanding, you won’t be able to understand or use other language features such as LINQ. This lesson will explain by example why we need generics and how to build your own should you need to.

* Arrays: Useful to manage small amounts of fixed-size data of the same type. An array or strings or integers are common examples. However, there are many times where you need a more flexible data structure that allows you to dynamically add and remove items at runtime.
* Non-generic Collections: Were designed to dynamically resize themselves as items are added and removed and address some of the limitations of the fixed size array. The problem with non-generic collections is that they typically operate on “Object” types and are therefore classified as loosely typed containers. While this mechanism provides more flexibility than a basic array, it introduces its own set of problems.
* What problem are generics trying to solve? A commonly used non-generic collection is the “ArrayList”. Because any type of object can be added to this collection, we can’t safely process individual items the same way. This can lead to unpredictable and random runtime errors because we can add any type of object to the collection at any point in time. If you EXPECT a certain type to exist in the collection, but you suddenly find the wrong type, chances are your code will not behave as expected and could throw a runtime exception. While there are times where we need a loosely typed container that can hold any type of item, but more often than not we need a way to ENSURE that all items in the collection are of the same type. Before “generics” were introduced, the only way to make a type-safe collection was to laboriously code it by hand. The samples “NonGenericSwap” and “NonGenericCarCollection” class demonstrates this process. It took me approximately 20 minutes to code and test. This might not seem like a big deal, but if you needed 10 or 15 such collections (representing tables in a database?), then you’d find yourself duplicating the same code over and over again, but simply replacing the type of object being stored in the collection. There has to be a better and quicker way!
* Introducing Generics: When you use generic collection classes, you rectify all the previous issues. Rather than having to build unique classes that can contain cars, cats, or any other object type that you’re dealing with, you can use a generic collection class and specify the type of object that the collection MUST have (hence, type-safe). When you see a generic item listed in any documentation, you will notice a pair of angled brackets with a letter or other token sandwiched within it. For example:

List<T> : Specifies a list that contains only objects of type “T”

When you create an instance of a generic class or structure, you specify the type parameter when you declare the variable and when you invoke the constructor. For example:

// This List<> can hold only Cat objects.

List<Cat> myCats = new List<Cat>();

// This List<> can hold only integers.

List<int> myInts = new List<int>();

* Creating your own generic type: While the generic types provided in the System.Collections.Generic namespace are extensive and will probably suit most of your needs, there are times where you might need to create your own generic type. To this end we will create a simple “GenericSwap” class that demonstrates the basics of how a generic type is created.

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

a = b;

b = temp;

// Compiler error!

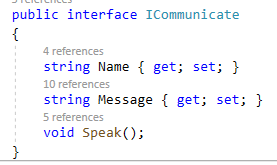
// a.Speak();

// b.Speak();

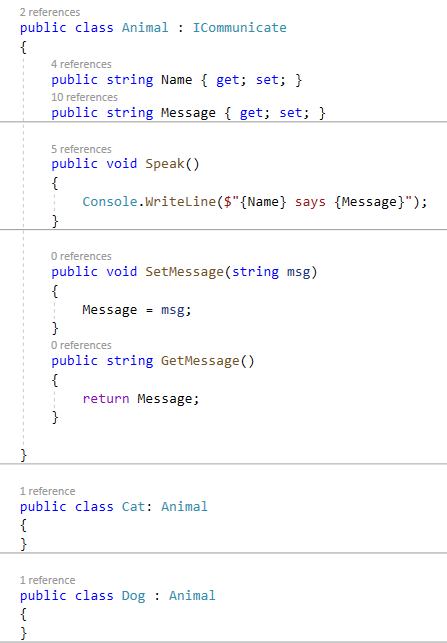
}

The generic type specifier <T> dictates the type that the code will act on. As such you can only do things with the parameters that are allowed to be performed by an object of type <T>.

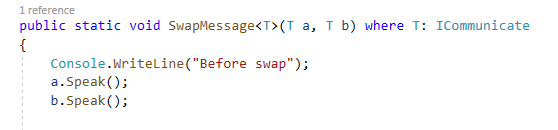
If the compiler doesn’t know what type <T> is then it can’t infer anything about that type other than that its some kind of value or reference type. Let’s say that for arguments sake that the objects that are being passed in as parameter a and b have a method called “Speak”. If we try to call this method we’ll get a compiler error. Why? Because the compiler cannot ASSUME anything about what type <T> you’ll be using in your code. You could specify that T is an integer or that it’s a boolean. Neither of these types contains a method called “Speak” and so that would be an error. However, pretend we have the following interface:



Now further assume that you have a concrete class that implements this interface, as in this example:

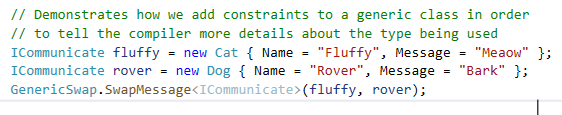


If we now used a Cat or a Dog in our call to swap, both of these objects DO have a method called “Speak” and SHOULD be ok to use. However, we still get a compiler error because the compiler does not have any GUARANTEE that you will use an object that is compliant. The question that now begs to be asked is “how can I tell the compiler that I GUARANTEE the objects I call swap with will be compliant”? The answer is with the “where” keyword. The “where” keyword provides the compiler with the metadata (type constraints) that it needs in order to determine more precisely the type of object you plan on using in this call. Let’s make a new method that provides one such constraint:



There are several different constraints that can be used with the “where“ keyword. In this particular case we specified an interface that the parameter type MUST support. We’ve indicated that type <T> MUST implement the interface ICommunicate.

Because both the Cat and Dog classes implement the ICommunicate interface, they can both be safely used in the call to SwapMessage:



The compiler will enforce this rule as well as any other “where” type constraints. This will GUARANTEE to the compiler that it will only be dealing with types that are compliant with the specified constraints. Since the compiler KNOWS that type ICommunicate implements method “Speak”, it is now safe to write code inside the generic method that makes use that, or ANY other method that is part of the interface.

There are several more “where” type constraints. They are:

where T : struct

where T : class

where T : new()

where T : NameOfBaseClass

where T : NameOfInterface

All of these can be used (some in combination with each other) to provide a great amount of detail to the compiler about the type <T> that is being used in the generic. It is left as an exercise for the student to come up with scenarios where each of these constraints would be required.