

CHAPTER 5



Yashica Mat 124G

Ghost Walker

UNDERSTANDING GENERICS

LEARNING OBJECTIVES

- *STATE THE BENEFITS OF USING GENERIC COLLECTION TYPES VS. NON-GENERIC COLLECTION TYPES*
- *USE GENERIC TYPE PARAMETERS TO CREATE GENERIC TYPES AND METHODS*
- *STATE THE PURPOSE OF GENERIC TYPE PARAMETERS AND GENERIC TYPE ARGUMENTS*
- *STATE THE LIMITATIONS OF UNBOUNDED TYPE PARAMETERS*
- *LIST FOUR DIFFERENT TYPES THAT CAN BE GENERIC TYPES*
- *LIST AND DESCRIBE SIX GENERIC TYPE CONSTRAINTS*
- *LIST AND DESCRIBE THE INTERFACES TARGETED BY GENERIC COLLECTION CLASSES*

INTRODUCTION

This chapter offers you a peek under the covers to reveal the inner workings of generic types and methods. Programmers unfamiliar with the .NET framework and with generic types specifically are initially bewildered by the confusing syntax used in the generic collection classes. What exactly does the ‘T’ represent in a `List<T>` collection class? Another question I hear frequently is “How does the `List<T>` class know how to manipulate different types when performing operations like sorting?” When you finish reading this chapter you’ll know the answer to these questions and many more.

I’ll start by showing you how to create generic types and methods using single and multiple generic, unbounded type parameters. Next, you’ll learn how to apply type constraints when defining generic types. Once you have a good understanding of generic types, I’ll explain the benefits of using generic types vs. non-generic types.

Fundamentally, this chapter prepares you for a formal encounter with generic collection types later in the book. Let’s get to work!

CREATING GENERIC TYPES

A *generic type* (class, structure, interface, or delegate) is one that’s declared with the help of one or more *type parameters*. A type parameter serves as a place holder which is ultimately replaced wherever it appears in the code by some specific type referred to as a *type argument*. You can think of a generic type as acting like a *type template*; the purpose of the template is to create new types as if stamping them from a mold. You can create generic types using single or multiple type parameters. In this section I will focus the discussion on the creation of generic classes. I will postpone the discussion of other generic types (interfaces, structs, and delegates) until later in the book.

Let’s start by creating a generic class that contains a single type parameter in its definition.

USING A SINGLE TYPE PARAMETER

Example 5.1 gives the code for a generic class that uses a single type parameter in its definition.

5.1 *SimpleGeneric.cs*

```
1      using System;
2
3      public class SimpleGeneric<T> {
4
5          public void PrintValue(T arg){
6              Console.WriteLine(arg);
7          }
8      }
```

Referring to example 5.1 — the `SimpleGeneric` class has one *type parameter* signified by the character `T` that appears within the angle brackets “< >”. You can use any identifier name to signify the type parameter, but in the .NET generic framework you’ll generally see single-character uppercase letters used for this purpose.

When an instance of `SimpleGeneric` is created, the *type argument* supplied in place of `T` is substituted for `T` wherever it appears in the code. In this example the character `T` appears in the parameter list of the `PrintValue()` method. Example 5.2 shows the `SimpleGeneric` class in action.

5.2 *MainApp.cs*

```
1      public class MainApp {
2          public static void Main(){
3              SimpleGeneric<string> sg_1 = new SimpleGeneric<string>();
4              sg_1.PrintValue("Hello World");
5          }
6      }
```

Referring to example 5.2 — an instance of `SimpleGeneric` is created on line 3 by supplying the `string` type as a type argument. This has the effect of creating a new type (e.g., `SimpleGeneric<string>`). The `string` type replaces `T` in the definition of the `PrintValue()` method (See example 5.1, line 5). Figure 5-1 shows the results of running this program.

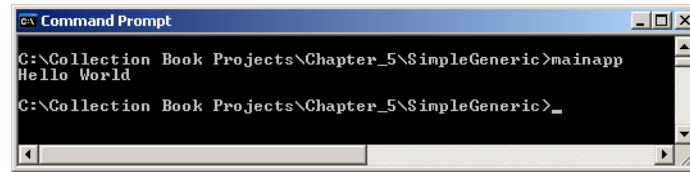


Figure 5-1: Results of Running Example 5.2

Using Multiple Type PARAMETERS

You can create generic types that use multiple type parameters. These type parameters can function as placeholders in methods, fields, properties, and other class member definitions. Example 5.3 gives the code for a class that uses two type parameters in its definition.

5.3 TwoParameterGeneric.cs

```

1      using System;
2
3      public class TwoParameterGeneric<T, U> {
4
5          //fields
6          private T _field1;
7          private U _field2;
8
9          //constructors
10         public TwoParameterGeneric(T arg1, U arg2){
11             _field1 = arg1;
12             _field2 = arg2;
13         }
14
15         private TwoParameterGeneric(){
16             // effectively disable the default constructor
17         }
18
19         //properties
20         public T PropertyOne {
21             get { return _field1; }
22             set { _field1 = value; }
23         }
24
25         public U PropertyTwo {
26             get { return _field2; }
27             set { _field2 = value; }
28         }
29
30         public U PrintValue(){
31             Console.WriteLine("T is a " + (_field1.GetType()).ToString() + " with value: " + _field1);
32             Console.WriteLine("U is a " + (_field2.GetType()).ToString() + " with value: " + _field2);
33             return _field2;
34         }
35     }

```

Referring to example 5.3 — the `TwoParameterGeneric` class declares two type parameters named `T` and `U` respectively. These type parameters appear throughout the code in field, constructor, property, and method definitions. Example 5.4 shows this class in action.

5.4 MainApp.cs

```

1      using System;
2
3      public class MainApp {
4          public static void Main(){
5              TwoParameterGeneric<string, int> tpg_1 = new TwoParameterGeneric<string, int>("Hello World", 4);
6              Console.WriteLine(tpg_1.PrintValue());
7              tpg_1.PropertyOne = "Second string";
8              tpg_1.PropertyTwo = 378;
9              Console.WriteLine("-----");
10             Console.WriteLine(tpg_1.PrintValue());
11         }
12     }

```

Referring to example 5.4 — an instance of the `TwoParameterGeneric` class is created on line 5 using the types `string` and `int` as type arguments. The string “Hello World” and the integer value 4 are passed as arguments to the constructor. Line 6 makes a call to the `PrintValue()` method and writes its return value to the console. Lines 7 and 8 dem-

onstrate the use of the properties and again on line 10 the `PrintValue()` method is called and its return value printed to the console. Figure 5-2 shows the results of running this program.

```

C:\Collection Book Projects\Chapter_5\TwoParameterGeneric>mainapp
T is a System.String with value: Hello World
U is a System.Int32 with value: 4
4
-----
T is a System.String with value: Second string
U is a System.Int32 with value: 378
378
C:\Collection Book Projects\Chapter_5\TwoParameterGeneric>_

```

Figure 5-2: Results of Running Example 5.4

UNBOUNDED TYPE PARAMETERS

The previous examples used *unbounded type parameters* in the definition of generic classes. There's not much you can do in the code with an unbounded type parameter. The reason why is that when presented with an unspecified interface for a type parameter, the compiler can only assume you mean to target the `System.Object` interface, which results in a very limited range of operations. That's why the examples I provide in this section do nothing more than print string values to the console via an object's `ToString()` method. Since every type (value, reference, delegate, etc.) ultimately extends from `System.Object`, I can safely write code in the body of my generic type examples that targets the `System.Object` interface. I can transcend this limitation by specifying a particular targeted interface via a *constraint*. You'll learn how to apply constraints to generic types later in this chapter, but first I want to show you how to create generic methods.

Quick Review

A *generic type* is one that's declared with the help of one or more *type parameters*. A type parameter serves as a place holder which will eventually be replaced wherever it appears in the code by some specific type referred to as a *type argument*. You can think of a generic type as acting like a *type template*; the purpose of the template is to create new types as if stamping them from a mold.

You can create generic types that use one or more type parameters. Type parameters can appear in the definition of any type member, including fields, constructors, properties, methods, etc.

In the absence of a type parameter *constraint*, the compiler assumes the targeted interface will be that of the `System.Object` class. An unconstrained type parameter is referred to as an *unbounded type parameter*.

CREATING GENERIC METHODS

A *generic method* is defined with the help of one or more type parameters that appear inside of angle brackets "<>". A generic method can appear in the definition of an ordinary, non-generic class or structure. That is, it's not a requirement for a class or structure to be generic for it to contain generic method definitions. Also, generic methods can define single or multiple type parameters. I say we fling ourselves into the deep end of the swimming pool and take a look at a generic method that uses multiple type parameters. Example 5.5 gives the code.

5.5 *GenericMethodDemo.cs*

```

1      using System;
2
3      public class GenericMethodDemo {
4
5          public T PrintValue<T, U>(T param1, U param2){
6              T local_var = param1;
7              Console.WriteLine("Parameter values are: param1 = " + param1 + " param2 = " + param2);
8              Console.WriteLine("Local variable value is: local_var = " + local_var);
9              return local_var;
10         }
11     }

```

Referring to example 5.5 — The `GenericMethodDemo` class is an ordinary, non-generic class. On line 5 it defines a generic method named `PrintValue<T, U>` that uses two type parameters `T` and `U` in its definition. This example demonstrates how the type parameters `T` and `U` can be used to specify the return type, method parameters, or local variables within the method. Example 5.6 demonstrates the use of the generic `PrintValue<T, U>()` method.

5.6 *MainApp.cs*

```

1      using System;
2
3      public class MainApp {
4          public static void Main(){
5              GenericMethodDemo gmd = new GenericMethodDemo();
6              gmd.PrintValue<string, int>("Hello World", 4); // explicit type parameters
7              Console.WriteLine("-----");
8              gmd.PrintValue(125.25, 62); //using generic type inference
9          }
10     }

```

Referring to example 5.6 — an instance of `GenericMethodDemo` is created on line 5. On line 6, the generic `PrintValue<T, U>()` method is called. Notice how the type arguments `string` and `int` are explicitly specified between the angle brackets “<>”. An alternative way to call a generic method is to let the compiler sort out the types via *generic type inference*. This concept is discussed in the following section.

GENERIC TYPE INFERENCE

Again referring to example 5.6 — line 8 demonstrates how the type arguments for the generic `PrintValue<T, U>()` can be sorted out automatically by the compiler via *generic type inference*. The types inferred by the this call to the `PrintValue<T, U>()` method are `double` and `int` respectively. Figure 5-3 shows the results of running example 5.6.

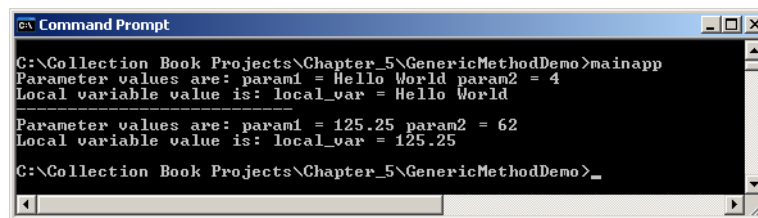


Figure 5-3: Results of Running Example 5.6

Quick Review

Generic methods use type parameters in their definition. A generic method definition may appear in the body of a non-generic class or structure.

There are two ways to call a generic method. 1) using explicit type parameters, or 2) letting the compiler figure out the types via generic type inference.

GENERIC TYPE CONSTRAINTS

When defining generic types and methods using unbounded type parameters, the compiler assumes the targeted base type is `System.Object`. This limits the range of valid operations you can perform on the subsequent type arguments you supply when you instantiate a generic type to those made public by the `System.Object`'s interface. To overcome this limitation you must specify a *generic type constraint* that instructs the compiler to limit the range of authorized type arguments to those that subscribe to certain conditions.

There are six types of generic type constraints: 1) *default constructor constraint*, 2) *reference type constraint*, 3) *value type constraint*, 4) *class derivation constraint*, 5) *interface implementation constraint*, and 6) *naked constraint*. Of the six, you'll find numbers 4 and 5, the class derivation and interface implementation constraints, most useful for creating your own generic types. I discuss all six constraints in greater detail below.

DEFAULT CONSTRUCTOR CONSTRAINT

The default constructor constraint instructs the compiler to limit the range of acceptable type arguments to those types that supply a default constructor. A default constructor is a public constructor that omits a parameter list (i.e., a parameterless constructor).

At first glance you may question the utility of this type of constraint, but if your generic type creates instances (objects) of the type arguments you supply, then those types will need to define a default constructor. An typical example of a class that creates objects is a factory class. Examples 5.7 through 5.9 give the code for a class named `MyClass` that implements a default constructor, a generic factory class named, appropriately enough, `GenericFactory<T>` where the type parameter `T` is constrained to types that implement a default constructor, and a class named `MainApp` that serves as a test driver.

5.7 *MyClass.cs*

```

1      public class MyClass {
2
3          //field
4          private string _field1;
5
6          //default constructor
7          public MyClass():this("Hello World"){ }
8
9          //overloaded constructor
10         public MyClass(string s){
11             _field1 = s;
12         }
13
14         //property
15         public string PropertyOne {
16             get { return _field1; }
17             set { _field1 = value; }
18         }
19     }

```

Referring to example 5.7 — `MyClass` contains one field, two constructors, and a property named `PropertyOne`. One of the constructors is a default constructor. `PropertyOne` is a read-write property. Example 5.8 gives the code for the `GenericFactory<T>` class.

5.8 *GenericFactory.cs*

```

1      using System;
2
3      public class GenericFactory<T> where T: new() {
4
5          private static GenericFactory<T> factory;
6
7          public static GenericFactory<T> Instance {
8              get { if (factory != null) {
9                  return factory;
10             } else {
11                 factory = new GenericFactory<T>();
12                 return factory;
13             }
14         }
15     }
16
17     public T NewObject(){
18         return new T();
19     }
20 }

```

Referring to example 5.8 — the `GenericFactory<T>` class implements both the singleton and factory software design patterns. It also applies the *default constructor constraint* to the type parameter `T`. Notice how the constraint is defined. The keyword `where` is used, followed by the parameter `T`, followed by a colon `:`. Following the colon the constraint `new ()` signifies the default constructor constraint.

Here's how it this class works. The `GenericFactory<T>` class defines one private static field named `factory`. It also defines one public static property named `Instance`. The `Instance` property is read-only. If the `factory` field is not null, meaning an instance of `GenericFactory<T>` has already been created, then a reference to the field is returned. If the `factory` field is null, a new instance of `GenericFactory<T>` is created and assigned to the `factory` field before it is returned.

The `NewObject()` method simply returns new references to objects of type `T`. Note that the default constructor is used to create objects of type `T`. (i.e., `T()`). Example 5.9 shows the `GenericFactory<T>` class in action.

5.9 MainApp.cs

```

1      using System;
2
3      public class MainApp {
4          public static void Main(){
5              MyClass mc1 = GenericFactory<MyClass>.Instance.NewObject();
6              MyClass mc2 = GenericFactory<MyClass>.Instance.NewObject();
7              Console.WriteLine(mc1.PropertyOne);
8              Console.WriteLine(mc2.PropertyOne);
9              Console.WriteLine("-----");
10             mc1.PropertyOne = "A slightly different message string...";
11             Console.WriteLine(mc1.PropertyOne);
12             Console.WriteLine(mc2.PropertyOne);
13         }
14     }

```

Referring to example 5.9 — the `GenericFactory<T>` class is used to create instances of `MyClass` as is indicated by using the type `MyClass` as a type parameter (i.e., `GenericFactory<MyClass>`). Since the `Instance` property is static, it's accessed via the type name. The type name in this case is `GenericFactory<MyClass>`. The `NewObject()` method, which is a non-static method, is called via the reference returned as a result of accessing the `GenericFactory<MyClass>.Instance` property. The remaining code in example 5.9 then exercises the two `MyClass` objects retrieved via the factory. Figure 5-4 shows the results of running this program.

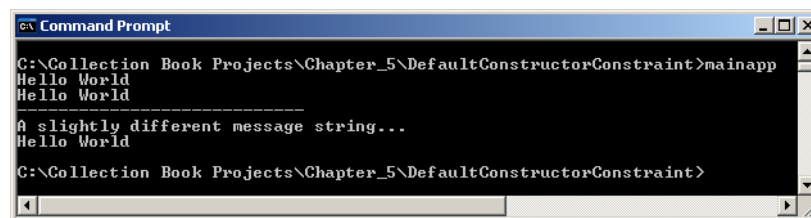


Figure 5-4: Results of Running Example 5.9

Note that the `GenericFactory<T>` class can be used to generate objects of any type that implements a default constructor. Remember that in C# if you don't explicitly define a default constructor or explicitly make the default constructor private or protected, the compiler will supply a default constructor. This is good enough to satisfy the default constructor constraint. Example 5.10 gives the code for a simple class that leaves the generation of a default constructor up to the compiler. Example 5.11 then uses the `GenericFactory<T>` class to create an object of this type.

5.10 SimpleClass.cs

```

1      public class SimpleClass {
2
3          private string _field1 = "Hello World";
4
5          // default constructor generated by compiler
6
7          public string PropertyOne {
8              get { return _field1; }
9              set { _field1 = value; }
10         }
11     }

```

Referring to example 5.10 — the definition of `SimpleClass` leaves the generation of the default constructor to the compiler. It defines one read-write property named `PropertyOne`.

5.11 MainApp.cs

```

1      using System;
2
3      public class MainApp {
4          public static void Main(){
5              SimpleClass sc = GenericFactory<SimpleClass>.Instance.NewObject();
6              Console.WriteLine(sc.PropertyOne);
7          }
8      }

```

Figure 5-5 shows the results of running this program.

REFERENCE/VALUE TYPE CONSTRAINTS

The purpose of the reference and value type constraints is to limit the range of valid type arguments to either reference types (classes) or value types (structures). You would use either of these constraints when, in order for the generic code to work properly, it needs to know if it's dealing with reference types or with value types. An example of

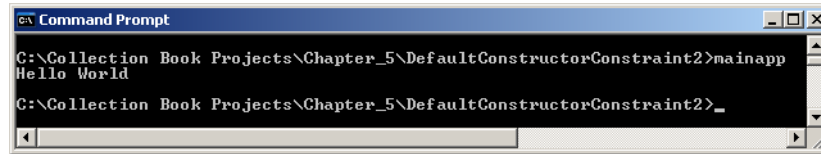


Figure 5-5: Results of Running Example 5.11

when this distinction would be important is when the semantics, or the meaning, of a particular expression would change according to whether the operands were classes or structures. (Classes having reference semantics and structures having value semantics.)

A case in point would be the difference in the behavior of *comparison or equality semantics* of reference types vs. value types. By default, reference types, when compared to each other for equality, perform a comparison (i.e., the `Object.Equals()` method) of one reference to another. Thus, two reference type objects with equal values will be found to be NOT equal if they are two distinct objects residing in two distinct memory locations. This is how comparisons work for reference types unless you explicitly override the `Object.Equals()` method to clarify the semantics of the equality comparison for your user-defined classes.

Conversely, a comparison of value type objects tests the contents of one against the contents of another, not their addresses.

REFERENCE TYPE CONSTRAINT

Let's take a look first at an example of a reference type constraint. To help with this example I shall enlist the aid of a class named `MyClass` given in example 5.12.

5.12 *MyClass.cs*

```

1      public class MyClass {
2          private int _field;
3
4          public MyClass():this(0){ }
5
6          public MyClass(int val){
7              _field = val;
8          }
9
10         public int Value {
11             get { return _field; }
12             set { _field = value; }
13         }
14     }

```

Referring to example 5.12 — `MyClass` defines one integer field named `_field`, two constructors, and one integer read-write property named `Value`. Fairly straight forward so far, nothing fancy. Example 5.13 gives the code for a generic class named `EqualityChecker<T>` where the type parameter `T` has been constrained to reference types.

5.13 *EqualityChecker.cs*

```

1      using System;
2
3      public class EqualityChecker<T> where T: class {
4
5          public bool CheckEquality(T a, T b){
6              bool result = a.Equals(b);
7              Console.WriteLine( result + " : {0} is " + (result?"":"not ") + "equal to {1}", a, b);
8              return ( a.Equals(b));
9          }
10     }

```

Referring to example 5.13 — the `EqualityChecker<T>` class applies a reference type constraint on the type parameter `T`. Note how the constraint is applied with use of the `class` keyword (where `T: class`). The `EqualityChecker<T>` class then goes on to define one method named `CheckEquality()` that takes two arguments of type `T` and performs an equality comparison of the two objects using the `System.Equals()` method on line 6. Note that the assumption made here is that all objects supplied, being class types, will subscribe to reference semantics, but this becomes hard to enforce since the `Object.Equals()` method can be overridden and thus its default behavior changed in derived classes. A case in point is the `String` class, where the `System.Object()` method is overridden to perform a comparison of one string's value (character sequence) against another's. Let's take a look at the `EqualityChecker<T>` class in action.

5.14 MainApp.cs

```

1      using System;
2
3      public class MainApp {
4          public static void Main(){
5              EqualityChecker<string> eq1 = new EqualityChecker<string>();
6              eq1.CheckEquality("Hello", "Hello");
7              eq1.CheckEquality("Hello", "World");
8              Console.WriteLine("-----");
9              EqualityChecker<MyClass> eq2 = new EqualityChecker<MyClass>();
10             eq2.CheckEquality(new MyClass(5), new MyClass(5));
11         }
12     }

```

Referring to example 5.14 — the `EqualityChecker<T>` class is instantiated first using the `string` type. On lines 6 and 7 the `CheckEquality()` method is called using string literals, first with two of the same value, next with two different values. On line 9 a new instance of `EqualityChecker<T>` is created using the `MyClass` type. On line 10 the `CheckEquality()` method is once again called with two instances of `MyClass` whose fields are initialized to the same value (5). Figure 5-6 shows the results of running this program.

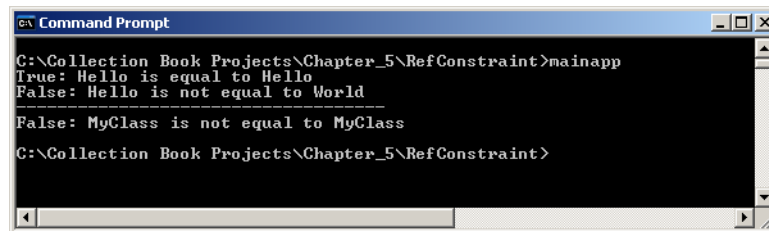


Figure 5-6: Results of Running Example 5.14

Referring to figure 5-6 — note the output for the string objects vs. the `MyClass` objects. Strings that contain identical character sequences are considered equal because that's the behavior defined by the `String` class's version of the overridden `Object.Equals()` method. Had I overridden the `System.Equals()` method in `MyClass`, I could have instructed it to behave in a similar fashion, comparing fields instead of addresses. But I didn't, and when writing generic code, you can't assume anything other than the worst case or common denominator.

VALUE TYPE CONSTRAINT

Let's look now at a modified version of the `EqualityChecker<T>` class, but this time I shall constrain the type parameters to value types. To help in this example I will use the `MyStruct` structure, the code for which is given in example 5.15.

5.15 MyStruct.cs

```

1      public struct MyStruct {
2
3          private int _field;
4
5          public MyStruct(int val){
6              _field = val;
7          }
8
9          public int Value {
10             get { return _field; }
11             set { _field = value; }
12         }
13     }
14

```

Referring to example 5.15 — the `MyStruct` structure defines one private integer field named `_field`, one constructor (explicit default constructors are not allowed in structures), and one public property named `Value`.

Example 5.16 gives the code for the slightly modified `EqualityChecker<T>` class that now has a *value type constraint* applied to the type parameter `T`.

5.16 EqualityChecker.cs (Mod 1)

```

1      using System;
2
3      public class EqualityChecker<T> where T: struct {
4
5          public bool CheckEquality(T a, T b){
6              bool result = a.Equals(b);
7              Console.WriteLine( result + " : {0} is " + (result?"":"not ") + "equal to {1}", a, b);

```

```

8         return ( a.Equals(b));
9     }
10 }

```

Referring to example 5.16 — with the exception of the value type constraint applied on line 3 using the `struct` keyword, this code remains otherwise unchanged from the previous version of `EqualityComparer<T>`. Example 5.17 puts this new version of `EqualityComparer<T>` through some paces.

5.17 *MainApp.cs*

```

1     using System;
2
3     public class MainApp {
4         public static void Main(){
5             EqualityChecker<int> eq1 = new EqualityChecker<int>();
6             eq1.CheckEquality(2, 2);
7             eq1.CheckEquality(3, 4);
8             Console.WriteLine("-----");
9             EqualityChecker<MyStruct> eq2 = new EqualityChecker<MyStruct>();
10            eq2.CheckEquality(new MyStruct(5), new MyStruct(5));
11        }
12    }

```

Referring to example 5.17 — on line 5 an instance of `EqualityComparer<T>`, `eq1`, is instantiated using an integer type argument. Lines 6 and 7 demonstrate calls to the `CheckEquality()` method using various integer literal values. On line 9 a second instance of `EqualityComparer<T>` is created using the `MyStruct` value type. On line 10 the `CheckEquality()` method is called using two fresh instances of `MyStruct` initialized to hold the same values. Figure 5-7 shows the results of running this program.

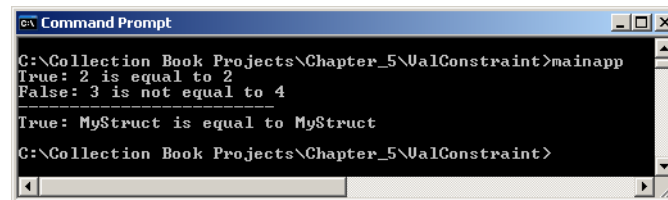


Figure 5-7: Results of Running Example 5.17

CLASS/INTERFACE DERIVATION/IMPLEMENTATION CONSTRAINTS

The class derivation and interface implementation constraints are by far the most useful generic type constraints. By targeting a specific type name, whether it be a class or an interface, you inform the compiler of your intent to use a specified set of operations on the type parameters. It is only by using the interface implementation or class derivation constraints that you can access overloaded operators. These constraints also let you take advantage of type substitution by specifying abstract base classes, interfaces, or a combination of both, and then substituting derived types in their place.

INTERFACE IMPLEMENTATION CONSTRAINT

The interface implementation constraint lets you constrain type parameters to a specific interface type. Example 5.18 gives the code for a generic class named `EqualityComparer<T>` that constrains the type parameter `T` to objects that implement the `IComparable` and `IComparable<T>` interfaces.

5.18 *EqualityComparer.cs*

```

1     using System;
2
3     public class EqualityChecker<T> where T: IComparable, IComparable<T> {
4
5         public bool CheckEquality(T a, T b){
6             bool return_val = false ;
7             int result = a.CompareTo(b);
8             if(result == 0){
9                 return_val = true;
10            }
11            Console.WriteLine(return_val + ": {0} is " + (return_val?"":"not ") + "equal to {1}", a, b);
12            return return_val;
13        }
14    }

```

Referring to example 5.18 — the `EqualityComparer<T>` class specifies two interface names in a comma-separated constraint list. On line 5, the `CheckEquality()` method takes two arguments of type `T` and compares one against the other using the `CompareTo()` method. (Types that realize the `Comparable` or `Comparable<T>` interfaces implement the `CompareTo()` method.) Line 11 prints the results of the comparison to the console.

Example 5.19 provides a short program that shows the `EqualityComparer<T>` class in action.

5.19 *MainApp.cs*

```

1      using System;
2
3      public class MainApp {
4          public static void Main(){
5              EqualityChecker<string> eq1 = new EqualityChecker<string>();
6              eq1.CheckEquality("Hello", "Hello");
7              Console.WriteLine("-----");
8              EqualityChecker<int> eq2 = new EqualityChecker<int>();
9              eq2.CheckEquality(2, 4);
10             Console.WriteLine("-----");
11         }
12     }

```

Referring to example 5.19 — an instance of `EqualityComparer<T>` is created on line 5 using the `string` type. On line 6, the `CheckEquality()` method compares two `string` literal values. On line 8 another instance of `EqualityComparer()` is instantiated but this time using an `integer` type argument. (Any type will work so long as it implements either the `Comparable` or `Comparable<T>` interfaces.) Figure 5-8 shows the results of running this program.

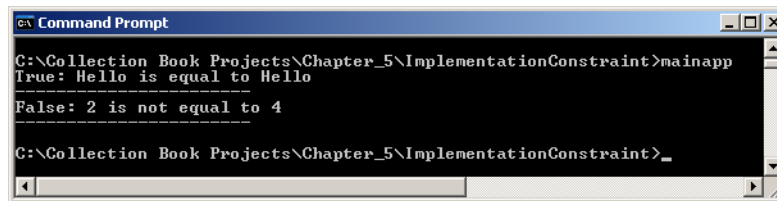


Figure 5-8: Results of Running Example 5.19

CLASS DERIVATION CONSTRAINT

The class derivation constraint lets you target a specific class type by name. Specifying a class name implies that subtypes of that class can be used as type arguments as well.

As I mentioned above, using either an interface implementation constraint or class derivation constraint is the only way to go if you wish to utilize overloaded operators on your type parameters. Example 5.20 gives the code for a class named `MyType` that overloads several operators. The `MyType` class is nothing more than a wrapper for an `integer` object.

5.20 *MyType.cs*

```

1      using System;
2
3      public class MyType {
4          private int _intField;
5
6          public int IntField {
7              get { return _intField; }
8              set { _intField = value; }
9          }
10
11         public MyType():this(5){
12             }
13
14         public MyType(int intField){
15             _intField = intField;
16         }
17
18         public static MyType operator +(MyType mt){
19             mt.IntField = (+mt.IntField);
20             return mt;
21         }
22
23         public static MyType operator -(MyType mt){
24             mt.IntField = (-mt.IntField);
25             return mt;
26         }

```

```

27
28     public static bool operator ! (MyType mt){
29         bool retVal = true;
30         if(mt.IntField >= 0){
31             retVal = false;
32         }
33         return retVal;
34     }
35
36     public static bool operator true(MyType mt){
37         return !mt;
38     }
39
40     public static bool operator false(MyType mt){
41         return !mt;
42     }
43
44     public static MyType operator ++ (MyType mt){
45         MyType result = new MyType(mt.IntField);
46         ++result.IntField;
47         return result;
48     }
49
50     public static MyType operator -- (MyType mt){
51         MyType result = new MyType(mt.IntField);
52         --result.IntField;
53         return result;
54     }
55
56     public static MyType operator +(MyType lhs, MyType rhs){
57         MyType result = new MyType(lhs.IntField);
58         result.IntField += rhs.IntField;
59         return result;
60     }
61
62     public static MyType operator -(MyType lhs, MyType rhs){
63         MyType result = new MyType(lhs.IntField);
64         result.IntField -= rhs.IntField;
65         return result;
66     }
67
68     public static MyType operator +(MyType lhs, int rhs){
69         MyType result = new MyType(lhs.IntField);
70         result.IntField += rhs;
71         return result;
72     }
73
74     public static MyType operator -(MyType lhs, int rhs){
75         MyType result = new MyType(lhs.IntField);
76         result.IntField -= rhs;
77         return result;
78     }
79
80     public static MyType operator *(MyType lhs, MyType rhs){
81         MyType result = new MyType(lhs.IntField);
82         result.IntField *= rhs.IntField;
83         return result;
84     }
85
86     public static MyType operator *(MyType lhs, int rhs){
87         MyType result = new MyType(lhs.IntField);
88         result.IntField *= rhs;
89         return result;
90     }
91
92     public static MyType operator /(MyType lhs, MyType rhs){
93         MyType result = new MyType(lhs.IntField);
94         result.IntField /= rhs.IntField;
95         return result;
96     }
97
98     public static MyType operator /(MyType lhs, int rhs){
99         MyType result = new MyType(lhs.IntField);
100        result.IntField /= rhs;
101        return result;
102    }
103
104    public static MyType operator &(MyType lhs, MyType rhs){
105        MyType result = new MyType(lhs.IntField);
106        result.IntField &= rhs.IntField;
107    }

```

```

108         return result;
109     }
110
111     public static MyType operator |(MyType lhs, MyType rhs){
112         MyType result = new MyType(lhs.IntField);
113         result.IntField |= rhs.IntField;
114         return result;
115     }
116
117     public static bool operator ==(MyType lhs, MyType rhs){
118         return lhs.IntField == rhs.IntField;
119     }
120
121     public static bool operator !=(MyType lhs, MyType rhs){
122         return lhs.IntField != rhs.IntField;
123     }
124
125     public static bool operator <(MyType lhs, MyType rhs){
126         return lhs.IntField < rhs.IntField;
127     }
128
129     public static bool operator >(MyType lhs, MyType rhs){
130         return lhs.IntField > rhs.IntField;
131     }
132
133     public static bool operator <=(MyType lhs, MyType rhs){
134         return lhs.IntField <= rhs.IntField;
135     }
136
137     public static bool operator >=(MyType lhs, MyType rhs){
138         return lhs.IntField >= rhs.IntField;
139     }
140
141     public static explicit operator int(MyType mt){
142         return mt.IntField;
143     }
144
145
146     // overridden System.Object methods
147     public override String ToString(){
148         return IntField.ToString();
149     }
150
151     public override bool Equals(object o){
152         if(o == null) return false;
153         if(!(o is MyType)) return false;
154         return this.ToString().Equals(o.ToString());
155     }
156
157     public override int GetHashCode(){
158         return this.ToString().GetHashCode();
159     }
160 } // end class definition

```

Referring to example 5.20 — the `MyType` class overloads most of the important operators. If you're unfamiliar with operator overloading please refer to chapter 22 in my book *C# For Artists: The Art, Philosophy, and Science of Object-Oriented Programming* (ISBN(13) 978-1-932504-07-1).

Example 5.21 gives the code for a generic class named, simply enough, `GenericType<T>`. (I must be running out of steam here!)

5.21 *GenericType.cs*

```

1     using System;
2
3     public class GenericType<T> where T: MyType {
4
5         public void PrintSum(T arg1, T arg2){
6             Console.WriteLine(arg1 + " + " + arg2 + " = " + (arg1 + arg2));
7         }
8     }

```

Referring to example 5.21 — The `GenericType<T>` class constrains the type parameter `T` to objects of type `MyType` or its subtypes. It defines one method named `PrintSum()` that takes two arguments of type `T` which, because of the derivation constraint, are guaranteed to be objects of type `MyType`. It then applies the binary addition operator '+' and prints the sum of the two object to the console. Figure 5-9 shows the results of running this program.

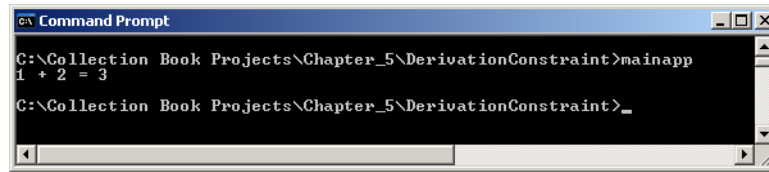


Figure 5-9: Results of Running Example 5.21

Naked CONSTRAINT

The naked constraint is used to define one type parameter in terms of another. To demonstrate the naked constraint I'll use two classes, one named `BaseClass` and one named `DerivedClass`, which, as you'll see, derives from `BaseClass`.

5.22 *BaseClass.cs*

```
1 public class BaseClass {
2
3     public virtual string InterfaceMethod(){
4         return "String returned from BaseClass";
5     }
6 }
```

Referring to example 5.22 — `BaseClass` defines one public virtual method named `InterfaceMethod()`. All this method does when called is return the string literal shown on line 4. Example 5.23 gives the code for `DerivedClass`.

5.23 *DerivedClass.cs*

```
1 public class DerivedClass : BaseClass {
2
3     public override string InterfaceMethod(){
4         return "String returned from Derived class method.";
5     }
6 }
```

Referring to example 5.23 — `DerivedClass` extends `BaseClass` and provides its own implementation of `InterfaceMethod()` by overriding the `BaseClass.InterfaceMethod()`. Example 5.24 gives the code for a class named `GenericClass<T>`.

5.24 *GenericClass.cs*

```
1 using System;
2
3 public class GenericClass<T> where T: BaseClass {
4
5     public void GenericMethod<U>(U arg) where U: T {
6         Console.WriteLine(arg.InterfaceMethod());
7     }
8 }
```

Referring to example 5.24 — the `GenericClass<T>` applies a derivation constraint to the type parameter `T` limiting the range of acceptable type arguments to type `BaseClass` and its derived types. On line 5 a generic method named `GenericMethod<U>(U arg)` is defined and a naked constraint is applied to the type parameter `U` that says, in effect, “limit `U` to the type specified by `T` or its subtypes.” Example 5.25 shows the `GenericClass<T>` in action.

5.25 *MainApp.cs*

```
1 using System;
2
3 public class MainApp {
4     public static void Main(){
5         GenericClass<BaseClass> gcl = new GenericClass<BaseClass>();
6         gcl.GenericMethod<BaseClass>(new BaseClass());
7         gcl.GenericMethod<BaseClass>(new DerivedClass());
8         gcl.GenericMethod<DerivedClass>(new DerivedClass());
9     }
10 }
```

Referring to example 5.25 — an instance of `GenericClass<T>` is created on line 5 using `BaseClass` as a type argument. Notice on lines 6 through 8 how three different versions of the `GenericMethod<U>` method are called. On line 6, `BaseClass` is used as a type argument and a new instance of `BaseClass` is created and used as a method argument; on line 7, `BaseClass` is used as a type argument and a new instance of `DerivedClass` is created and used as a method argument; and finally, on line 8, `DerivedClass` is used as a type argument and a new instance of `DerivedType` is created and used as a method argument. Figure 5-10 shows the results of running this program.

```

C:\Collection Book Projects\Chapter_5\NakedConstraint>mainapp
String returned from BaseClass
String returned from Derived class method.
String returned from Derived class method.
C:\Collection Book Projects\Chapter_5\NakedConstraint>

```

Figure 5-10: Results of Running Example 5.25

Limited Utility of the Naked Constraint

In example 5.24 I combined the *derivation constraint* with the *naked constraint* for demonstration purposes. Had I not specified the derivation constraint limiting the acceptable type arguments to those of type `BaseClass` and its derivatives, I would have been unable to access the `InterfaceMethod()` method in the code and would have been limited to the interface published by the `System.Object` class. With this in mind, I could have simply done away with the naked constraint as applied to the `GenericMethod<U>()` method and rewritten the `GenericClass<T>` with only the derivation constraint to the same effect. The simplified code for an alternative version of `GenericClass<T>` appears in example 5.26.

5.26 *GenericClass.cs (Mod 1)*

```

1      using System;
2
3      public class GenericClass<T> where T: BaseClass {
4
5          public void GenericMethod(T arg){
6              Console.WriteLine(arg.InterfaceMethod());
7          }
8      }

```

Referring to example 5.26 — the naked constraint has been removed for the `GenericMethod()` declaration. This simplifies the code with no effect on functionality. The derivation constraint limits the range of type arguments to `BaseClass` and its derived types, but as you have learned, the derivation and interface constraints are the most useful to you anyway.

Example 5.27 gives the modified `MainApp` class.

5.27 *MainApp.cs (Mod 1)*

```

1      using System;
2
3      public class MainApp {
4          public static void Main(){
5              GenericClass<BaseClass> gc1 = new GenericClass<BaseClass>();
6              gc1.GenericMethod(new BaseClass());
7              gc1.GenericMethod(new DerivedClass());
8          }
9      }

```

Figure 5-11 shows the results of running this program.

```

C:\Collection Book Projects\Chapter_5\NakedConstraint_Mod1>mainapp
String returned from BaseClass
String returned from Derived class method.
C:\Collection Book Projects\Chapter_5\NakedConstraint_Mod1>

```

Figure 5-11: Results of Running Example 5.27

CONSTRAINT SUMMARY TABLE

Table 5-1 lists and summarizes the constraints presented in this section. It offers recommendations for their use and briefly describes issues you need to consider when applying constraints.

Constraint	Form	Implementation
Default Constructor Constraint	<code><T> where T: new() { ... }</code>	Use when code needs to create objects of type T
Reference Type Constraint	<code><T> where T: class { ... }</code>	Use when code needs to know if reference semantics apply to objects of type T.
Value Type Constraint	<code><T> where T: struct { ... }</code>	Use when code needs to know if value type semantics apply to objects of type T.
Interface Implementation Constraint	<code><T> where T: <i>interface_name</i> { ... }</code>	Use when code needs to know that objects of type T implement the interface as indicated by <i>interface_name</i> . This is a very useful constraint because it lets you access all operations defined by the specified interface.
Class Derivation Constraint	<code><T> where T: <i>class_name</i> { ... }</code>	Use when code needs to know that objects of type T from the class indicated by <i>class_name</i> . This is a very useful constraint because it lets you access all operations defined by the specified class interface.
Naked Constraint	<code><T, U> where T: U { ... }</code>	Specifies objects of type T in terms of U. Effectively limits objects of T to those of type U and its derivatives. Limited usefulness because the only operations available on objects of type T are those specified by System.Object. Favor the use of either the derivation constraint or implementation constraint.

Table 5-1: Constraint Summary Table

Quick Review

Use generic type constraints to limit the range of acceptable type arguments in generic types. There are six type constraints: 1. Default constructor constraint, 2. Reference type constraint, 3. Value type constraint, 4. Interface implementation constraint, 5. Class derivation constraint, and 6. Naked constraint. Of the six, the interface implementation and class derivation constraints are most helpful.

BENEFITS OF USING GENERIC TYPES

The use of generic types offers several important benefits over non-generic types including increased type safety, saved space, improved performance, less work, and improved code quality. I discuss each of these benefits in more detail below.

INCREASED Type Safety

The use of generics reduces and in many cases eliminates the need for the programmer to perform type checks and casting operations. When you create a generic type, type checks on the type parameters and type arguments are performed at compile time, eliminating runtime type errors.

As you learned in this chapter, the enforcement of type safety imposes limits on what you can get away with when you create a generic type. For example, you can't apply operators willy-nilly to unbounded type parameters because the compiler can't guarantee the type argument eventually supplied to instantiate the generic type will implement those operators.

GENERICS SAVE SPACE

The key rationale for generic types derives from the benefit of writing a *general-purpose routine* that can be used in *multiple contexts*. This saves space because it eliminates the need for multiple code assemblies, each one perhaps created to manipulate different types of objects using the same, repeated code pattern.

GENERICS IMPROVE PERFORMANCE

Generics types have the potential to improve code performance, especially in compute-intensive code segments where the boxing and unboxing of value-types would incur overhead. To illustrate this point I have written a short program that adds 1 million integers to a non-generic `ArrayList` and a generic `List<int>` and then sorts each list, recording the time it takes to complete the sort operation on each collection. Example

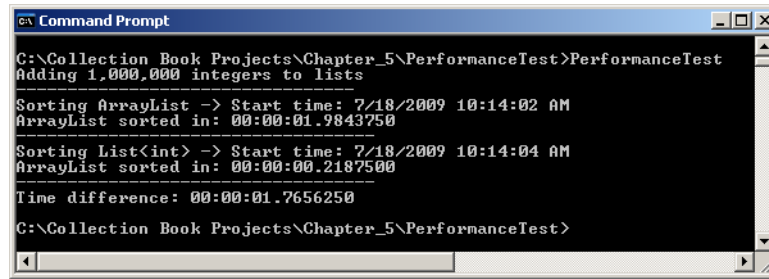
5.28 *PerformanceTest.cs*

```

1      using System;
2      using System.Collections;
3      using System.Collections.Generic;
4
5      public class PerformanceTestOne {
6          public static void Main(){
7              ArrayList list = new ArrayList();
8              List<int> generic_list = new List<int>();
9              int NUMBER = 1000000;
10
11              Console.WriteLine("Adding {0:N0} integers to lists", NUMBER);
12              Console.WriteLine("-----");
13              Random random = new Random();
14              for(int i=0; i<NUMBER; i++){
15                  int temp = random.Next();
16                  list.Add(temp);
17                  generic_list.Add(temp);
18              }
19              DateTime start = DateTime.Now;
20              Console.WriteLine("Sorting ArrayList -> Start time: " + start);
21              list.Sort();
22              TimeSpan array_list_elapsed_time = (DateTime.Now - start);
23              Console.WriteLine("ArrayList sorted in: " + array_list_elapsed_time);
24
25              Console.WriteLine("-----");
26
27              start = DateTime.Now;
28              Console.WriteLine("Sorting List<int> -> Start time: " + start);
29              generic_list.Sort();
30              TimeSpan list_elapsed_time = (DateTime.Now - start);
31              Console.WriteLine("ArrayList sorted in: " + list_elapsed_time);
32
33              Console.WriteLine("-----");
34              Console.WriteLine("Time difference: " + (array_list_elapsed_time - list_elapsed_time));
35          }
36      }
37

```

Referring to example 5.28 — this program compares the performance of a non-generic collection against that of a generic collection in the sorting of integers. The non-generic `ArrayList` will incur a boxing and unboxing performance penalty because integer value-types must be “boxed” into objects with being inserted into the `ArrayList`, (since it is object-based) and unboxed with performing the sort comparisons. Figure 5-12 shows the results of running this program.



```
C:\Collection Book Projects\Chapter_5\PerformanceTest>PerformanceTest
Adding 1,000,000 integers to lists
-----
Sorting ArrayList -> Start time: 7/18/2009 10:14:02 AM
ArrayList sorted in: 00:00:01.9843750
-----
Sorting List<int> -> Start time: 7/18/2009 10:14:04 AM
ArrayList sorted in: 00:00:00.2187500
-----
Time difference: 00:00:01.7656250
C:\Collection Book Projects\Chapter_5\PerformanceTest>
```

Figure 5-12: Results of Running Example 5.28

Referring to figure 5-12 — the old-school `ArrayList` collection took 1.98 seconds to sort while the generic `List<int>` collection took only .218 seconds to sort. That's an improvement of approximately 90%. Your times will most certainly vary from mine but you should see similar results.

GENERICs ELIMINATE WORK AND IMPROVE CODE QUALITY

The use of generic types, that is, the creation of general purpose code that works with multiple data types, can potentially save you a lot of work and improve code quality. You save time and eliminate redundant work by writing code that can be reused in different contexts. For example, code that sorts `Strings` can sort numeric data as well. (The ordering behavior is implemented in the targeted data type, as you'll learn in subsequent chapters.)

Code that can be reused in different contexts tends to have to bugs worked out. This can really be applied not only to generics, but to the whole .NET framework. The .NET framework is not without its issues and problems, but the more the code is used and tested, the more bugs are discovered and fixed in subsequent releases.

Quick Review

The use of generic types offers several important benefits over non-generic types including increased type safety, saved space, improved performance, less work, and improved code quality.

SUMMARY

A *generic type* is one that's declared with the help of one or more *type parameters*. A type parameter serves as a place holder which will eventually be replaced wherever it appears in the code by some specific type referred to as a *type argument*. You can think of a generic type as acting like a *type template*; the purpose of the template is to create new types as if stamping them from a mold.

You can create generic types that use one or more type parameters. Type parameters can appear in the definition of any type member, including fields, constructors, properties, methods, etc.

In the absence of a type parameter *constraint*, the compiler assumes the targeted interface will be that of the `System.Object` class. An unconstrained type parameter is referred to as an *unbounded type parameter*.

Use generic type constraints to limit the range of acceptable type arguments in generic types. There are six type constraints: 1. *Default constructor constraint*, 2. *Reference type constraint*, 3. *Value type constraint*, 4. *Interface implementation constraint*, 5. *Class derivation constraint*, and 6. *Naked constraint*. Of the six, the interface implementation and class derivation constraints are most helpful.

The use of generic types offers several important benefits over non-generic types including increased type safety, saved space, improved performance, less work, and improved code quality.

REFERENCES

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [<http://www.ecma-international.org/publications/standards/Ecma-335.htm>]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [<http://www.ecma-international.org/publications/standards/Ecma-334.htm>]

Microsoft Developer Network (MSDN) [<http://www.msdn.com>]

NOTES
