|  |
| --- |
|  |
| C# Generics |
| Calculators & Containers |
|  |
|  |

# 

# C# Generics

## Objectives

After completing this lab, you should understand how to:

* Use a generic collection class in the System.Collections.Generics namespace
* Use generics when building classes and methods.

## Overview

In this lab we’ll be building and consuming generic types. One of the objectives is to build a simple inversion of control (IoC) container.

What’s an IoC container?

We don’t want to derail the objective of this lab, which is to learn about generics, but it turns out that IoC containers make heavy use of generics, so building one from scratch is a useful exercise (although for a real application, you’ll want to use one of the many that are already available). In short: a container helps you manage and decouple dependencies in a software application. For example, let’s say you are writing a class that needs to use a second component to record audit information. One way to get hold of the second component is to instantiate it directly.

SqlLogger logger = new SqlLogger();

In some scenarios, however, the above approach would tie your code too tightly to the implementation of a logging component that writes to SQL Server. What happens if you want to unit test the code without setting up a database? What happens if you want to switch the audit message destination throughout the application?

A decoupled approach could rely on dependency injection and an IoC container. Instead of direct instantiation of a concrete type, like SqlLogger, your class could allow an IoC container to provide you with an object that knows how to record the audit messages. Your code won’t know the ultimate destination of the messages, and the code becomes more flexible and testable. We’ll see how this works soon, but for more on dependency injection and containers, read “[Tame Your Software Dependencies for More Flexible Apps](http://msdn.microsoft.com/en-us/magazine/cc337885.aspx)” by James Kovacs.

## Part 1 – Generic Collections

In this first section we’ll be taking a program that uses a non-generic Hashtable collection and modifying it to use a Dictionary<TKey, TValue>.

1. Create a new project in Visual Studio using the **File -> New -> Project** menu commands.
2. In the New Project dialog box, select the **Visual C# -> Console Application** template. Name the project **Calculator**, and select the **Before directory** of this lab as the Location before pressing OK.
3. In the Solution Explorer Window (**Ctrl+Alt+L**), right-click the Calculator project and select **Add -> Class**. Use **Container.cs** as the name and click Add.
4. **Replace** the code in Container.cs with the following:

using System;

using System.Collections;

namespace Calculator

{

public static class Container

{

public static void Register(Type contract, Type implementation)

{

\_map.Add(contract, implementation);

}

public static object Resolve(Type contract)

{

return Activator.CreateInstance(\_map[contract] as Type);

}

static Hashtable \_map = new Hashtable();

}

}

This is the beginning of a simple container. The class will allow us to configure the container by associating a concrete type with an interface using the Register method. Other components can then invoke the Resolve method to instantiate an object that implements a specific interace.

1. In the Solution Explorer Window, right-click the Calculator project and select **Add -> New Item**. Select **Interface** from the list of item templates, and use ILogger as the name.
2. **Replace** the code in ILogger.cs with the following:

namespace Calculator

{

interface ILogger

{

void Log(string message);

}

}

1. In the Solution Explorer Window, right-click the Calculator project and select **Add -> Class.**  Use ConsoleLogger as the class name.
2. Replace the code in ConsoleLogger.cs with the following:

using System;

namespace Calculator

{

class ConsoleLogger : ILogger

{

public void Log(string message)

{

Console.WriteLine(message);

}

}

}

1. In the Solution Explorer Window, right-click the Calculator project and select **Add -> Class.**  Use Calculator as the class name.
2. Replace the code in Calculator.cs with the following:

using System;

namespace Calculator

{

class Calculator

{

public void Add(int value)

{

\_logger.Log("Adding " + value);

Result += value;

}

public void Subtract(int value)

{

\_logger.Log("Subtracting " + value);

Result -= value;

}

public int Result { get; set; }

ILogger \_logger = Container.Resolve(typeof(ILogger))

as ILogger;

}

}

At this point we have the basic pieces in place. See if you can write the code in Program.cs to configure the container, instantiate a Calculator, and perform some basic arithmetic operations (which we will show in the next step).

1. In the Solution Explorer window, double-click on **Program.cs**.
2. Change the **Main** method to look like the following.

static void Main(string[] args)

{

Container.Register(typeof(ILogger),

typeof(ConsoleLogger));

Calculator calc = new Calculator();

calc.Add(3);

calc.Subtract(2);

Console.WriteLine("The result is {0}", calc.Result);

Console.ReadLine();

}

1. Press **F5** to run the program and test the results.

The Container class gives us some decoupling between the Calculator and the ConsoleLogger. The Calculator doesn’t need to instantiate a specific logger, in fact the Calculator doesn’t even know the ConsoleLogger exists – it only asks the Container for something that implements ILogger and works against the ILogger interface. It’s up to the application to configure the Container with the specific logger it wants to use at runtime. Many containers allow you to do this configuration using an XML file so you can modify the behavior of a program without recompiling. Notice the container is usng the Activator class in .NET to dynamically instantiate an object using a Type parameter.

Although the decoupling is nice, there are a few problems. Do you remember how many type coercions are needed in the code? There are 2 – and we can eliminate both using generics. With generics, we can also ensure that the data structure used inside the container will only store a mapping of Type objects to Type objects. Currently this data structure is a hashtable, meaning we could accidently store a Type and an Int32 or a String and a Type – which isn’t what we want to do.

Let’s start by changing our Container from using a Hashtable to a generic Dictionary.

1. Open **Container.cs**
2. Change the type of the **\_map field** from Hashtable to **Dictionary<Type,Type>**. You’ll also need to add a using for the **System.Collections.Generic** namespace.
3. **Remove** the using statement for the **System.Collections** namespace.
4. **Remove** the type cohercion inside the **Resolve** method (the *as Type expression*).
5. Press **F5** to make sure the program still builds and executes correctly.

The type cohercion was required when using the Hashtable, because the C# compiler cannot tell what type of object we’ll retrieve from the Hashtable. Using a Dictionary<Type,Type>, the C# compiler knows we are pulling out a Type object. The full source for the Container class now looks like the following.

using System;

using System.Collections.Generic;

namespace Calculator

{

public static class Container

{

public static void Register(Type contract, Type implementation)

{

\_map.Add(contract, implementation);

}

public static object Resolve(Type contract)

{

return Activator.CreateInstance(\_map[contract]);

}

static Dictionary<Type, Type> \_map =

new Dictionary<Type, Type>();

}

}

Some people don’t like the verbosity of generics (and some people have a general disdain for < and > characters). In some scenarios you can use inheritance to avoid generic type parameters and create more readable code (as demonstrated in the following steps).

1. In the Solution Explorer Window, right-click the Calculator project and select **Add -> Class.**  Use TypeMap as the class name.
2. Replace the code in TypeMap.cs with the following:

using System;

using System.Collections.Generic;

namespace Calculator

{

public class TypeMap : Dictionary<Type,Type>

{

}

}

1. Re-open the **Container.cs** file. **Replace Dictionary<Type,Type>** with **TypeMap** in the \_map declaration.

static TypeMap \_map = new TypeMap();

1. Press F5 and test the result.

We can use generics to improve the Container class even more. In the next step we will introduce some generic methods.

## Part II – Generic Methods

The container class can still use some improvement – particularly the Resolve method. Currently anyone who uses the Resolve method has to cast or coerce the return type since Resolve currently returns System.Object. Using generics we can return an object with he exact type requested by the caller.

Generally speaking, anytime you are passing a System.Type argument as a method parameter, you might have the opportunity to use a generic type parameter instead. In many cases the generic type parameter is the preffered approach because it makes the code type safe.

See if you can change the Resolve method to use a generic type parameter and remove the “as” expression in Calculator.cs. The following steps will show you how.

1. Open the **Container.cs** file.
2. Add a generic type parameter named **T** to the **Resolve method**.
3. Use the new type parameter as the **return type** for Resolve.
4. **Remove** the **method paramter** (contract) from the Resolve method.
5. Whever the contract parameter was used, you can now use **typeof(T)** in it’s place.
6. Finally, use the type parameter to **cast** the return value of Activator.CreateInstance.
7. The Container code should now look like the following:

using System;

using System.Collections.Generic;

namespace Calculator

{

public static class Container

{

public static void Register(Type contract, Type implementation)

{

\_map.Add(contract, implementation);

}

**public static T Resolve<T>()**

**{**

**return (T)Activator.CreateInstance(\_map[typeof(T)]);**

**}**

static TypeMap \_map = new TypeMap();

}

}

1. Open **Calculator.cs**.
2. **Change** the line of code invoking Container.Resolver to **pass a generic type parameter**, and **remove the “as”** expression (as shown below).

ILogger \_logger = Container.Resolve<ILogger>();

Hopefully you’ll see the above code as an improvement over the previous version. Before we leave, however, we can go back to the Container class and make some additional improvements. We want to use generic type parameters with our Register method. See if you can make the change yourself, or follow along with the steps below.

1. Open **Container.cs**.
2. **Add** a new Register method that uses **two generic type parameters** to create a type mapping entry.

You’ll now have the following two Register methods in Container and clients will have an option to invoke one or the other, depending on which they prefer. The method you just added should look like the second method below.

public static void Register(Type contract, Type implementation)

{

\_map.Add(contract, implementation);

}

**public static void Register<TContract, TImplementation>()**

**{**

**\_map.Add(typeof(TContract), typeof(TImplementation));**

**}**

1. Open **Program.cs**
2. Change the registration code to use the new Register method.

Container.Register<ILogger, ConsoleLogger>();

In this scenario, generic type parameters allow us to register types without using a typeof operator.

1. Press F5 to run the application and test the result.

## Part III – Generics and Events

In this section we’ll add an event to the Calculator class that takes advantage of the EventHandler<T> class in .NET. These events will fire if the Result the calculator is tracking goes outside the range of 0-100.

The first step in raising a custom event Is to define a class that will carry any custom information we want to publish during an event. In the case of the calculator, we want to publish the current value (Result) the calculator is holding.

1. In the Solution Explorer Window (**Ctrl+Alt+L**), right-click the Calculator project and select **Add -> Class**. Use **RangeWarningEventArgs.cs** as the name and click Add.
2. **Replace** the code in **RangeWarningEventArgs.cs** with the following:

using System;

namespace Calculator

{

class RangeWarningEventArgs

{

public int Value { get; set; }

}

}

If you’ve worked with events before, you might notice a subtle problem with the code. If you don’t spot it, don’t worry! We’ll come back later and fix it.

Note that before generics came along, we’d have to also define a custom delegate that uses the new event args class we just created - the code would look like the following (**don’t add this code**).

public delegate void RangeWarningHandler(object sender,

RangeWarningEventArgs args);

However, with EventHandler<t> we can skip the above code.

1. Open **Calculator.cs**
2. **Add** the following **event** definition to the Calculator class.

public event EventHandler<RangeWarningEventArgs> RangeWarning;

1. **Build** the solution (Shift+Ctrl+B). You should see a compiler error.

The compiler is telling us we can’t use RangeWarningEventArgs as a type parameter to EventHandler<T>. Can you figure out why?

In .NET, events arguments should always derive from the System.EventArgs class. To help enforce this convention, EventHandler<T> has a constraint that says T should be, or derive from, System.EventArgs. For our code to compile, we need to comply with this constraint.

1. Open **RangeWarningEventArgs.cs**.
2. Use **EventsArgs** as the base class for the RangeWarningEventsArgs class.

using System;

namespace Calculator

{

class RangeWarningEventArgs **: EventArgs**

{

public int Value { get; set; }

}

}

1. **Build** the solution and make sure there are no errors left.

Now we will add the code to raise this event. See if you can add this code yourself, or follow along with these steps:

1. **Open Calculator.cs**
2. Add a **CheckResult method** to Calculator with the following code.

void CheckResult()

{

if (Result < 0 || Result > 100)

{

EventHandler<RangeWarningEventArgs> handler = RangeWarning;

if (handler != null)

{

RangeWarningEventArgs args = new RangeWarningEventArgs();

args.Value = Result;

handler(this, args);

}

}

}

1. In the **Add** and **Subtract** methods of Calculator, invoke CheckResult after performing the requested mathematical operation.

public void Add(int value)

{

\_logger.Log("Adding " + value);

Result += value;

**CheckResult();**

}

public void Subtract(int value)

{

\_logger.Log("Subtracting " + value);

Result -= value;

**CheckResult();**

}

Finally, we’ll modify the main program to catch and process this event.

1. Open **Program.cs**
2. Inside of Main, after the calculator is instantiated, **subscribe to** the **RangeWarning** event on the calc object.

calc.RangeWarning +=

new EventHandler<RangeWarningEventArgs>(calc\_RangeWarning);

Notice we can once again use the generic EventHandler<T> to describe a method that will handle the event.

Note also that once you type the += in the above code that you can press the TAB key twice to let Visual Studio generate the rest of the line for the event subscription, and the event handling method, too.

1. In the event handler, write a message to the console to **print** the current Calculator value.

static void calc\_RangeWarning(object sender,

RangeWarningEventArgs e)

{

Console.WriteLine(

"The calculator value ({0}) is dangerous",

e.Value);

}

1. Finally, modify the Main method and force the Calculator to raise the warning event. Complete code is shown below.

using System;

namespace Calculator

{

class Program

{

static void Main(string[] args)

{

Container.Register<ILogger, ConsoleLogger>();

Calculator calc = new Calculator();

calc.RangeWarning +=

new EventHandler<RangeWarningEventArgs>

(calc\_RangeWarning);

calc.Add(97);

calc.Add(4);

calc.Subtract(2);

Console.WriteLine("The result is {0}", calc.Result);

Console.ReadLine();

}

static void calc\_RangeWarning(object sender,

RangeWarningEventArgs e)

{

Console.WriteLine(

"The calculator value ({0}) is dangerous",

e.Value);

}

}

}

1. **Run** the program (F5) and test the result.

## Conclusion

In this lab we’ve used generic collections and also implemented logic using generic methods and generic events. At this point you should understand not just the syntax of generic type parameters, but also how they can benefit your code through type safety and in some cases – readabiliy. If you want some additional challenges with generics, implement a new generic collection that is currently not available on the .NET framework, like a [prioritized queue](http://en.wikipedia.org/wiki/Priority_queue) or a [deque](http://en.wikipedia.org/wiki/Double-ended_queue).