SODV 2202

Object Oriented Programming

Module 3

Generics

Module 3: Generics

Introduction

What you will learn and why it is important:

One of the major goals in software development is to write code that is reusable. Every piece of code that gets written should be tested before it is used. Once some code has been tested and improved, developers can safely reuse the code because it has been proven to be reliable. Inheritance provided one mechanism for reusing code by making derived classes that include all the features of a base class. Generics are a different mechanism for reusing code, in which the developer writes a template for a solution that can operate across different types of data.

In this module you will learn how to create generic types and methods. You will also learn about many of the useful generic types provided in standard libraries.

Learning Outcomes

By the end of this module you will:

Explain how generic code works and develop reusable code using generic types and methods.

Learning Objectives

Here is what you will do in each lesson to achieve the learning outcome:

1. Generic Types and Methods
    - Create generic data types.
    - Implement generic methods.
2. Constraints
    - Apply constraints to generic code.
3. Collections
    - Use standard collections to store and access data.

Performance Evaluation

To show you have learned the material, here is what you will be asked to complete:

The topics in this module are needed to complete Assignment 1.

Lesson 3.1: Generic Types and Methods

In this lesson you will:

- Create generic data types.
- Implement generic methods.

Resources:

- Textbook: *C# 7.0 in a Nutshell*, Chapter 3, Sections "Generic Types", "Why Ge "Generic Methods", "Declaring Type Parameters", "typeof and Unbound Gene and "The default Generic Value", Pages 122 – 126.
- Video: Youtube.com, Brackeys, 15. How to program in C# - GENERICS – Tutori

Generics allow you to write code that is independent of the type of data it is operating on. C# is a ***type-safe*** language, meaning that variables are defined with a particular type and cannot refer to variables of a different type without some sort of conversion taking place. A ***generic type*** allows you to create a class (or structure) where you do not commit to the types of data used by that class when you write it. For example, let's say we need a class to define a 2D point, but sometimes we need the coordinates to be integers, and sometimes we need them to be floats:

```
public class Point<T>
{
    public T x { get; set; }
    public T y { get; set; }

    public override string ToString()
    {
        return String.Format("Point(x: {0}, y: {1})", x, y);
    }
}

class Program
{
    static void Main(string[] args)
    {
```

In this example, *T* is the ***type parameter***, and it is used as a placeholder for any type we want the user to be able to replace.

Generics can have multiple type parameters, like in the following example which defines a tuple. A tuple is a simple data structure that pairs 2 or more pieces of data.

It is also possible to make individual methods generic. To do this, place the *type parameters* after the function name, but before the normal parameter list. When calling a ***generic method***, the type parameters may be omitted if they can be deduced from the parameter list.

```
class Program
{
    public static void Swap<T>(ref T value1, ref T value2)
    {
        T temp = value1;
        value1 = value2;
        value2 = temp;
    }

    static void Main(string[] args)
```

This example creates a temporary variable of the generic type. If the variable were not initialized it would get a default value. The default value of a type can be accessed directly using the ***default*** keyword.

```
class Program
{
    public static void PrintDefault<T>()
    {
        Console.WriteLine(default(T));
    }
```

Lesson 3.2: Constraints

In this lesson you will:

- Apply constraints to generic code.

Resources:

- Textbook: *C# 7.0 in a Nutshell*, Chapter 3, Section "Generic Constraints", Page
- Video: Youtube.com, Jeremy Clark, JeremyBytes – C# Generics – Part 5: Gener

Generics are often used to define reusable data structures. If you attempt to do much with a generic type besides store it, the instruction may not be possible for all types that could be used. ***Constraints*** allow you to restrict a generic type or method to be only types that meet certain requirements. For example, the ***new()*** constraint restricts the type to reference types with default constructors:

```
public class CreateLater<T> where T: new()
{
    private T _instance;
    public T instance
    {
        get
        {
            if (_instance == null)
                _instance = new T();
            return _instance;
        }
    }
}

class PrintOnCreate
{
    public int value;
    public PrintOnCreate()
    {
        Console.WriteLine("Constructor");
    }
}

class Program
{
    static void Main(string[] args)
```

The following table lists the different constraints that can be applied to a generic:

Constraint	*Description*
where T : struct	The type must be a value type.
where T : class	The type must be a reference type.
where T : unmanaged	The type must not be a reference and must not contain any reference types.
where T : new()	The type must have a public default constructor.
where T : <base class>	The type must be or derive from a specific class.
where T : <interface>	The type must be or implement a specific class.
where T1 : T2	The type must be or derive from the type given in another type argument.

Lesson 3.3: Collections

In this lesson you will:

- Use standard collections to store and access data.

Resources:

- Textbook: *C# 7.0 in a Nutshell*, Chapter 7, Section "Generic Collections", Page
- Video: Lynda.com, C#: Interfaces and Generics, Section 3 "C# Generics": https://www.lynda.com/C-tutorials/C-Interfaces-Generics/388134-2.html?

The .NET Framework provides a large assortment of useful data structures in its ***Collections*** library. You've already worked with ***Arrays***, which are fixed-length groups of elements identified by their array ***index***. TODO more info?

List<T> is a resizable array with many additional helper functions included. You can ***Add***, ***Insert***, or ***Remove*** objects from the *List*. Among many other features, *Lists* have functions to ***Sort*** and ***Search***. You can access any element in the *List* by index using the *[] operator*.

As an alternative to *List<T>*, you may encounter ***ArrayList***. *ArrayList* is not generic, it stores reference to the base *Object* class, allowing it to store objects of different types. The generic implementation of *List<T>* will be significantly faster when working with value type objects, and it can capture much the same functionality as *ArrayList* by making a *List<Object>*.

LinkedList<T> is a generic doubly-linked list. In a linked list, each object stores a reference to the next object in the sequence. This means that linked lists do not have random access like arrays do, they must be read in order. However, linked lists are also significantly faster than arrays at adding and removing data from the sequence.

The .NET framework also provides standard generic implementations of ***Queue<T>*** and ***Stack<T>***, as well as non-generic ***Queue*** and ***Stack*** classes that store references to *Objects*.

HashSet<T> is a collection that can be quickly searched to see if it contains a particular value. ***SortedSet<T>*** is similar except that it also maintains the set in a sorted order when iterating over it.

Dictionaries are collections that store ***key-value*** pairs. Values are retrieved by providing a key to look up. The .NET Framework provides a large number of different dictionaries that vary on their internal data structure and specific features. In general ***Dictionary<K, V>*** is fast and efficient for most uses.