Using Collections and Building Generic Types

Lab A: Using Collections

Exercise 1: Optimizing a Method by Caching Data

Task 1: Open the Collections solution

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Open the Collections solution in the E:\Labfiles\Lab 12\Lab A\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 12\Lab A\Ex1\Starter** folder, click **Collections.sln**, and then click **Open**.

Task 2: Modify the Gauss class to implement the memoization pattern

1. In the TestHarness project, display the MainWindow.xaml window:
   - In Solution Explorer, expand the **TestHarness** project, and then double-click **MainWindow.xaml**.

   The MainWindow window implements a simple test harness to enable you to test the method that you will use to perform Gaussian elimination. This is a Windows® Presentation Foundation (WPF) application that enables a user to enter the coefficients for four simultaneous equations that consist of four variables (w, x, y, and z). It then uses Gaussian elimination to find a solution for these equations. The results are displayed in the lower part of the screen.
2. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

3. In the task list, locate the **TODO - Add a static Hashtable** task, and then double-click this task.

   This task is located in the GaussianElimination project, in the **Gauss** class.

4. At the top of the Gauss.cs file, at the end of the list of **using** statements, add a statement to bring the **System.Collections** namespace into scope.

   Your code should resemble the following code example.

```
...

using System.Text;
using System.Collections;
namespace GaussianElimination

...
```

5. Remove the comment, and then add code to define a static **Hashtable** object named **results**.

   Your code should resemble the following code example.

```
...

public static class Gauss
{
    static Hashtable results;

    /// <summary>

    ...
}

...
```

6. At the beginning of the **SolveGaussian** method, before the statements that create a deep copy of the parameters, add code to ensure that the **results Hashtable** object is initialized. Create a new instance of this object if it is currently **null**.

   Your code should resemble the following code example.

```
...

public static double[] SolveGaussian
   (double[,] coefficients, double[] rhs)
{
```

```
    if (results == null)
    {
        results = new Hashtable();
    }

    ...
}
 ...
```

7. Add code to generate a hash key that is based on the method parameters by performing the following tasks:

    a. Define a new **StringBuilder** object named **hashString**.

    b. Iterate through the **coefficients** array, and append each value in the array to the **hashString StringBuilder** object.

    c. Iterate through the **rhs** array, and append each value in the array to the **hashString StringBuilder** object.

    d. Define a new **string** object named **hashValue**, and initialize it to the value that the **hashString.ToString** method returns.

Hint: This procedure generates a hash key by simply concatenating the values that are passed into the method. You can use more advanced hashing algorithms to generate better hashes. The **System.Security.Cryptography** namespace includes many classes that you can use to implement hashing.

Your code should resemble the following code example.

```
...

public static double[] SolveGaussian
   (double[,] coefficients, double[] rhs)
{
    ...

    StringBuilder hashString = new StringBuilder();

    foreach (double coefficient in coefficients)
    {
        hashString.Append(coefficient);
    }

    foreach (double value in rhs)
    {
```

```
        hashString.Append(value);
    }

    string hashValue = hashString.ToString();

    ...
}
...
```

8. Add code to check whether the **results** object already contains a key that has the value in the **hashValue** string. If it does, return the value that is stored in the **Hashtable** collection class that corresponds to the **hashValue** key. If the **results** object does not contain the **hashValue** key, the method should use the existing logic in the method to perform the calculation.

Hint: A **HashTable** object stores and returns values as objects. You must cast the value that is returned from a **HashTable** object to the appropriate type before you work with it. In this case, cast the returned value to an array of **double** values.

Your code should resemble the following code example.

```
...
public static double[] SolveGaussian
   (double[,] coefficients, double[] rhs)
{
    ...
    string hashValue = hashString.ToString();

    if (results.Contains(hashValue))
    {
        return (double[])results[hashValue];
    }

    else
    {
        // Make a deep copy of the parameters

        ...
        return rhsCopy;
    }

}
...
```

9. In the task list, locate the **TODO - Store the result of the calculation in the Hashtable** task, and then double-click this task.

   This task is located near the end of the **SolveGaussian** method.

10. Remove the comment, and then add code to the method to store the **rhsCopy** array in the **HashTable** object, specifying the **hashValue** object as the key.

    Your code should resemble the following code example.

```
...
public static double[] SolveGaussian
   (double[,] coefficients, double[] rhs)
{
    ...
    else{
        ...
        System.Threading.Thread.Sleep(5000);
        results.Add(hashValue, rhsCopy);
        return rhsCopy;
    }
}
...
```

Task 3: Test the solution

1. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
2. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
3. In the MainWindow window, enter the following equations, and then click **Solve**:

Note: Enter a value of zero in the corresponding text if no value is specified for w, x, y, or z in the equations below.

- $2w + x - y + z = 8$
- $-3w - x + 2y + z = -11$
- $-2w + x - 2y = -3$
- $3w - x + 2y - 2z = -5$

Observe that the operation takes approximately five seconds to complete.

4. Verify that the following results are displayed:
   - w = 4
   - x = −17
   - y = −11
   - z = 6
5. Modify the third equation to match the following equation, and then click **Solve** again:
   - −2w + x − 2y **+ 3z** = −3

   Observe that this operation also takes approximately five seconds to complete.
6. Verify that the following results are displayed:
   - w = −2
   - x = 25
   - y = 7
   - z = −6
7. Undo the change to the third equation so that all of the equations match those in Step 3, and then click **Solve**. Observe that this time, the operation takes much less time to complete because it uses the solution that was generated earlier.
8. Close the application, and then close Visual Studio:
   - In Visual Studio, on the **File** menu, click **Exit**.

Lab B: Building Generic Types

Exercise 1: Defining a Generic Interface

Task 1: Open the GenericTypes solution

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Import the code snippets from the E:\Labfiles\Lab 12\Lab B\Snippets folder:
   a. In Visual Studio, on the **Tools** menu, click **Code Snippets Manager**.
   b. In the **Code Snippets Manager** dialog box, in the **Language** drop-down list box, select **Visual C#**.
   c. Click **Add**.
   d. In the **Code Snippets Directory** dialog box, move to the **E:\Labfiles\Lab 12\Lab B\Snippets** folder, and then click **Select Folder**.
   e. In the **Code Snippets Manager** dialog box, click **OK**.
4. Open the GenericTypes solution in the E:\Labfiles\Lab 12\Lab B\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 12\Lab B\Ex1\Starter** folder, click **GenericTypes.sln**, and then click **Open**.

Task 2: Define the generic IBinaryTree interface

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

2. In the task list, locate the **TODO – Define the IBinaryTree interface** task, and then double-click this task. This task is located in the IBinaryTree.cs file.

3. In the **BinaryTree** namespace, define a new generic public interface named **IBinaryTree**. This interface should take a single type parameter named **TItem**. Specify that the type parameter must implement the generic **IComparable** interface.

   Your code should resemble the following code example.

```
...
namespace BinaryTree
{
    public interface IBinaryTree<TItem>
        where TItem : IComparable<TItem>
    {
    }
}
```

4. In the **IBinaryTree** interface, define the following public methods:

   a. An **Add** method, which takes a **TItem** object named **newItem** as a parameter and does not return a value.

   b. A **Remove** method, which takes a **TItem** object named **itemToRemove** as a parameter and does not return a value.

   c. A **WalkTree** method, which takes no parameters and does not return a value.

   Your code should resemble the following code example.

```
...
namespace BinaryTree
{
    public interface IBinaryTree<TItem>
        where TItem : IComparable<TItem>
    {
        void Add(TItem newItem);

        void Remove(TItem itemToRemove);

        void WalkTree();
    }
}
```

5. Build the solution and correct any errors:

   - On the **Build** menu, click **Build Solution**. Correct any errors.

Exercise 2: Implementing a Generic Interface

Task 1: Open the GenericTypes solution

Note: Perform this task only if you have not been able to complete Exercise 1. If you have defined the **IBinaryTree** interface successfully, proceed directly to **Task 2: Create the Tree class**.

- Open the GenericTypes solution in the E:\Labfiles\Lab 12\Lab B\Ex2\Starter folder:
  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
  b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 12\Lab B\Ex2\Starter** folder, click **GenericTypes.sln**, and then click **Open**.

Task 2: Create the Tree class

1. In the BinaryTree project, add a new class named **Tree**:
   a. In Solution Explorer, right-click the **BinaryTree** project, point to **Add**, and then click **Class**.
   b. In the **Add New Item - BinaryTree** dialog box, in the **Name** box, type **Tree** and then click **Add**.
2. Modify the **Tree** class definition. This class should be a public generic class that takes a single type parameter called **TItem** and implements the **IBinaryTree** interface. The **TItem** type parameter must implement the generic **IComparable** interface.

   Your code should resemble the following code example.

```
...
namespace BinaryTree
{
    public class Tree<TItem> : IBinaryTree<TItem>
        where TItem : IComparable<TItem>
    {
    }
}
...
```

3. Add the following automatic properties to the **Tree** class:

a. A **TItem** property named **NodeData**.

b. A generic **Tree<TItem>** property named **LeftTree**.

c. A generic **Tree<TItem>** property named **RightTree**.

Your code should resemble the following code example.

```
...
public class Tree<TItem> : IBinaryTree<TItem>
    where TItem : IComparable<TItem>
{
    public TItem NodeData { get; set; }

    public Tree<TItem> LeftTree { get; set; }

    public Tree<TItem> RightTree { get; set; }
}

...
```

4. Add a public constructor to the **Tree** class. The constructor should take a single **TItem** parameter called *nodeValue*. The constructor should initialize the **NodeData** member by using the *nodeValue* parameter, and then set the **LeftTree** and **RightTree** members to **null**.

   Your code should resemble the following code example.

```
...
public class Tree<TItem> : IBinaryTree<TItem>
    where TItem : IComparable<TItem>
{
    ...

    public Tree(TItem nodeValue)
    {
        this.NodeData = nodeValue;
        this.LeftTree = null;
        this.RightTree = null;
    }
}

...
```

5. After the constructor, define a method called **Add**. This method should take a **TItem** object as a parameter, but not return a value.

   Your code should resemble the following code example.

```
public class Tree<TItem> : IBinaryTree<TItem>
    where TItem : IComparable<TItem>
{
    ...
    public Tree(TItem nodeValue)
    {
        this.NodeData = nodeValue;
        this.LeftTree = null;
        this.RightTree = null;
    }

    public void Add(TItem newItem)
    {
    }
}
...
```

6. In the **Add** method, add code to insert the **newItem** object into the tree in the appropriate place by performing the following tasks:
    a. Compare the value of the **newItem** object with the value of the **NodeData** property. Both items implement the **IComparable** interface, so use the **CompareTo** method of the **NodeData** property. The **CompareTo** method returns zero if both items have the same value, a positive value if the value of the **NodeData** property is greater than the value of the **newItem** object, and a negative value if the value of the **NodeData** property is less than the value of the **newItem** object.
    b. If the value of the **newItem** object is less than the value of the **NodeData** property, perform the following actions to insert a **newItem** object into the left subtree:
        i. If the **LeftTree** property is **null**, initialize it and pass the **newItem** object to the constructor.
        ii. If the **LeftTree** property is not **null**, recursively call the **Add** method of the **LeftTree** property and pass the **newItem** object as the parameter.
    c. If the value of the **newItem** object is greater than or equal to the value of the **NodeData** property, perform the following actions to insert the **newItem** object into the right subtree:
        i. If the **RightTree** property is **null**, initialize it and pass the value of the **newItem** object to the constructor.

ii. If the **RightTree** property is not **null**, recursively call the **Add** method of the **RightTree** property and pass the **newItem** object as the parameter.

Your code should resemble the following code example.

```
public void Add(TItem newItem)
{
    TItem currentNodeValue = this.NodeData;

    // Check if the item should be inserted in the left tree.
    if (currentNodeValue.CompareTo(newItem) > 0)
    {
        // Is the left tree null?
        if (this.LeftTree == null)
        {
            this.LeftTree = new Tree<TItem>(newItem);
        }
        else // Call the Add method recursively.
        {
            this.LeftTree.Add(newItem);
        }
    }
    else // Insert in the right tree.
    {
        // Is the right tree null?
        if (this.RightTree == null)
        {
            this.RightTree = new Tree<TItem>(newItem);
        }
        else // Call the Add method recursively.
        {
            this.RightTree.Add(newItem);
        }
    }
}
```

7. After the **Add** method, add another public method called **WalkTree** that does not take any parameters and does not return a value.

   Your code should resemble the following code example.

```
public class Tree<TItem> : IBinaryTree<TItem>
    where TItem : IComparable<TItem>
{
    ...

    public void Add(TItem newItem)
    {
```

```
        ...
    }

    public void WalkTree()
    {
    }
}
...
```

8. In the **WalkTree** method, add code that visits each node in the tree in order and displays the value that each node holds by performing the following tasks:

    a. If the value of the **LeftTree** property is not **null**, recursively call the **WalkTree** method on the **LeftTree** property.

    b. Display the value of the **NodeData** property to the console by using a **Console.WriteLine** statement.

    c. If the value of the **RightTree** property is not **null**, recursively call the **WalkTree** method on the **RightTree** property.

    Your code should resemble the following code example.

```
public void WalkTree()
{
    // Recursive descent of the left tree.

    if (this.LeftTree != null)
    {
        this.LeftTree.WalkTree();
    }

    Console.WriteLine(this.NodeData.ToString());

    // Recursive descent of the right tree.

    if (this.RightTree != null)
    {
        this.RightTree.WalkTree();
    }

}
```

9. After the **WalkTree** method, add the **Remove** method to delete a value from the tree, as the following code example shows. It is not necessary for you to fully understand how this method works, so you can either type this code manually or use the Mod12Remove code snippet.

```
public void Remove(TItem itemToRemove)
{

    // Cannot remove null.
    if (itemToRemove == null)
    {
        return;
    }

    // Check if the item could be in the left tree.
    if (this.NodeData.CompareTo(itemToRemove) > 0
        && this.LeftTree != null)
    {
        // Check the left tree.
        // Check 2 levels down the tree - cannot remove
        // 'this', only the LeftTree or RightTree properties.
        if (this.LeftTree.NodeData.CompareTo(itemToRemove) == 0)
        {
            // The LeftTree property has no children - set the
            // LeftTree property to null.
            if (this.LeftTree.LeftTree == null
                && this.LeftTree.RightTree == null)
            {
                this.LeftTree = null;
            }
            else // Remove LeftTree.
            {
                RemoveNodeWithChildren(this.LeftTree);
            }
        }
        else
        {
            // Keep looking - call the Remove method recursively.
            this.LeftTree.Remove(itemToRemove);
        }
    }

    // Check if the item could be in the right tree.?
    if (this.NodeData.CompareTo(itemToRemove) < 0
        && this.RightTree != null)
    {
        // Check the right tree.

        // Check 2 levels down the tree - cannot remove
        // 'this', only the LeftTree or RightTree properties.
        if (this.RightTree.NodeData.CompareTo(itemToRemove) == 0)
        {
            // The RightTree property has no children - set the
            // RightTree property to null.
```

```
            if (this.RightTree.LeftTree == null
                && this.RightTree.RightTree == null)
            {
                this.RightTree = null;
            }

            else // Remove the RightTree.
            {
                RemoveNodeWithChildren(this.RightTree);
            }

        }

        else
        {
            // Keep looking - call the Remove method recursively.
            this.RightTree.Remove(itemToRemove);
        }
    }

    // This will only apply at the root node.
    if (this.NodeData.CompareTo(itemToRemove) == 0)
    {
        // No children - do nothing, a tree must have at least
        // one node.
        if (this.LeftTree == null && this.RightTree == null)
        {
            return;
        }

        else // The root node has children.
        {
            RemoveNodeWithChildren(this);
        }

    }
}
```

- To use the code snippet, type **Mod12Remove** and then press the TAB key twice.

10. After the **Remove** method, add the **RemoveNodeWithChildren** method to remove a node that contains children from the tree, as the following code example shows. This method is called by the **Remove** method. Again, it is not necessary for you to understand how this code works, so you can either type this code manually or use the Mod12RemoveNodeWithChildren code snippet.

```
private void RemoveNodeWithChildren(Tree<TItem> node)
{
    // Check whether the node has children.
    if (node.LeftTree == null && node.RightTree == null)
    {
        throw new ArgumentException("Node has no children");
    }

    // The tree node has only one child - replace the
    // tree node with its child node.
    if (node.LeftTree == null ^ node.RightTree == null)
    {
        if (node.LeftTree == null)
        {
            node.CopyNodeToThis(node.RightTree);
        }
        else
        {
            node.CopyNodeToThis(node.LeftTree);
        }
    }
    else
    // The tree node has two children - replace the tree node's value
    // with its "in order successor" node value and then remove the
    // in order successor node.
    {
        // Find the in order successor - the leftmost descendant of
        // its RightTree node.
        Tree<TItem> successor = GetLeftMostDescendant(node.RightTree);

        // Copy the node value from the in order successor.
        node.NodeData = successor.NodeData;

        // Remove the in order successor node.
        if (node.RightTree.RightTree == null &&
            node.RightTree.LeftTree == null)
        {
            node.RightTree = null; // The successor node had no
                                   // children.
        }
        else
        {
            node.RightTree.Remove(successor.NodeData);
        }
    }
}
```

- To use the code snippet, type **Mod12RemoveNodeWithChildren** and then press the TAB key twice.

11. After the **RemoveNodeWithChildren** method, add the **CopyNodeToThis** method, as the following code example shows. The **RemoveNodeWithChildren** method calls this method to copy another node's property values into the current node. You can either type this code manually or use the Mod12CopyNodeToThis code snippet.

```
private void CopyNodeToThis(Tree<TItem> node)
{
    this.NodeData = node.NodeData;
    this.LeftTree = node.LeftTree;
    this.RightTree = node.RightTree;
}
```

- To use the code snippet, type **Mod12CopyNodeToThis** and then press the TAB key twice.

12. After the **CopyNodeToThis** method, add the **GetLeftMostDescendant** method, as the following code example shows. The **RemoveNodeWithChildren** method also calls this method to retrieve the leftmost descendant of a tree node. You can either type this code manually or use the Mod12GetLeftMostDescendant code snippet.

```
private Tree<TItem> GetLeftMostDescendant(Tree<TItem> node)
{
    while (node.LeftTree != null)
    {
        node = node.LeftTree;
    }
    return node;
}
```

- To use the code snippet, type **Mod12GetLeftMostDescendant** and then press the TAB key twice.

13. Build the solution and correct any errors:

- On the **Build** menu, click **Build Solution**. Correct any errors.

Exercise 3: Implementing a Test Harness for the BinaryTree Project

Task 1: Open the GenericTypes solution

Note: Perform this task only if you have not been able to complete Exercise 2. If you have defined the **IBinaryTree** interface and built the **Tree** class successfully, proceed directly to **Task 3: Complete the test harness**.

- Open the GenericTypes solution in the E:\Labfiles\Lab 12\Lab B\Ex3\Starter folder:
  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
  b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 12\Lab B \Ex3\Starter** folder, click **GenericTypes.sln**, and then click **Open**.

Task 2: Import the TestHarness project

Note: Perform this task only if you have completed Exercise 2 successfully.

1. Import the TestHarness project in the E:\Labfiles\Lab 12\Lab B\Ex3\Starter \TestHarness folder into the GenericTypes solution:
   a. In Visual Studio, in Solution Explorer, right-click **Solution 'GenericTypes' (1 Project)**, point to **Add**, and then click **Existing Project**.
   b. In the **Add Existing Project** dialog box, move to the **E:\Labfiles\Lab 12 \Lab B\Ex3\Starter\TestHarness** folder, click **TestHarness.csproj**, and then click **Open**.
2. In the TestHarness project, update the reference to the BinaryTree project:
   a. In Solution Explorer, in the TestHarness project, expand **References**, right-click **BinaryTree**, and then click **Remove**.
   b. Right-click **References**, and then click **Add Reference**.
   c. In the **Add Reference** dialog box, on the **Projects** tab, click **BinaryTree**, and then click **OK**.

3. Set the TestHarness project as the startup project:
    - In Solution Explorer, right-click **TestHarness**, and then click **Set as Startup Project**.

Task 3: Complete the test harness

1. Open the Program.cs file:
    - In Solution Explorer, in the TestHarness project, double-click **Program.cs**.
2. In the **Main** method, add code to instantiate a new **IBinaryTree** object named **tree**, using **int** as the type parameter. Pass the value **5** to the constructor. This code creates a new binary tree of integers and adds an initial node that contains the value **5**.

    Your code should resemble the following code example.

```
...

static void Main(string[] args)
{
    IBinaryTree<int> tree = new Tree<int>(5);
}

...
```

3. Add code to the **Main** method to add the following values to the tree, in the following order:
    a. **1**
    b. **4**
    c. **7**
    d. **3**
    e. **4**

    Your code should resemble the following code example.

```
...

static void Main(string[] args)
{
    IBinaryTree<int> tree = new Tree<int>(5);
    tree.Add(1);
    tree.Add(4);
```

```
    tree.Add(7);
    tree.Add(3);
    tree.Add(4);
}
...
```

4. Add code to the **Main** method to perform the following actions:

    a. Print the message "Current Tree: " to the console, and then invoke the **WalkTree** method on the **tree** object.

    b. Print the message "Add 15" to the console, and then add the value **15** to the tree.

    c. Print the message "Current Tree: " to the console, and then invoke the **WalkTree** method on the **tree** object.

    d. Print the message "Remove 5" to the console, and then remove the value **5** from the tree.

    e. Print the message "Current Tree: " to the console, and then invoke the **WalkTree** method on the **tree** object.

    f. Pause at the end of the method until ENTER is pressed.

    Your code should resemble the following code example.

```
...
static void Main(string[] args)
{
    ...
    Console.WriteLine("Current Tree: ");
    tree.WalkTree();
    Console.WriteLine("Add 15");
    tree.Add(15);
    Console.WriteLine("Current Tree: ");
    tree.WalkTree();
    Console.WriteLine("Remove 5");
    tree.Remove(5);
    Console.WriteLine("Current Tree: ");
    tree.WalkTree();
    Console.ReadLine();
}...
```

5. Build the solution and correct any errors:

    - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 4: Test the BinaryTree project

1. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
2. Verify that the output in the console window resembles the following code example. Note that the data in the binary tree is sorted and is displayed in ascending order.

```
Current Tree:
1
3
4
4
5
7
Add 15
Current Tree:
1
3
4
4
5
7
15
Remove 5
Current Tree:
1
3
4
4
7
15
```

3. Press ENTER to close the console window, and then return to Visual Studio.

Exercise 4: Implementing a Generic Method

Task 1: Open the GenericTypes solution

- Open the GenericTypes solution in the E:\Labfiles\Lab 12\Lab B\Ex4\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 12\Lab B \Ex4\Starter** folder, click **GenericTypes.sln**, and then click **Open**.

Note: The GenericTypes solution in the Ex4 folder is functionally the same as the code that you completed in Exercise 3. However, it includes an updated task list and a new test project to enable you to complete this exercise.

Task 2: Create the BuildTree method

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. In the task list, locate the **TODO - Add the BuildTree generic method** task, and then double-click this task. This task is located at the end of the **Tree** class.
3. Remove the **TODO - Add the BuildTree generic method** comment, and then add a public static generic method named **BuildTree** to the **Tree** class. The type parameter for the method should be called **TreeItem**, and the method should return a generic **Tree<TreeItem>** object. The **TreeItem** type parameter must represent a type that implements the generic **IComparable** interface.

   The method should take two parameters: a **TreeItem** object called **nodeValue** and a **params** array of **TreeItem** objects called **values**.

   Your code should resemble the following code example.

```
public static Tree<TreeItem> BuildTree<TreeItem>
    (TreeItem nodeValue, params TreeItem[] values)
    where TreeItem : IComparable<TreeItem>
{

}
```

4. In the **BuildTree** method, add code to construct a new **Tree** object by using the data that is passed in as the parameters by performing the following actions:
   a. Define a new **Tree** object named **tree** by using the **TreeItem** type parameter, and initialize the new **Tree** object by using the **nodeValue** parameter.

b. Iterate through the **values** array, and add each value in the array to the **tree** object.

c. Return the **tree** object at the end of the method.

Your code should resemble the following code example.

```
public static Tree<TreeItem> BuildTree<TreeItem>
    (TreeItem nodeValue, params TreeItem[] values)
    where TreeItem : IComparable<TreeItem>
{

    Tree<TreeItem> tree = new Tree<TreeItem>(nodeValue);

    foreach (TreeItem item in values)
    {
        tree.Add(item);
    }
    return tree;

}
```

Task 3: Modify the test harness to use the BuildTree method

1. In the task list, locate the **TODO - Modify the test harness to use the BuildTree method** task, and then double-click this task.

   This task is located in the **Main** method of the Program.cs class file in the TestHarness project.

2. In the **Main** method, remove the existing code that instantiates the **tree** object and adds the first five values to the tree. Replace this code with a statement that calls the **BuildTree** method to create a new **Tree** object named **tree**, based on the integer type, with the following integer values:

   a. **1**

   b. **4**

   c. **7**

   d. **3**

   e. **4**

   f. **5**

   Your code should resemble the following code example.

```
static void Main(string[] args)
{
    IBinaryTree<int> tree =
        Tree<int>.BuildTree<int>(1, new int[]{4, 7, 3, 4, 5});
    Console.WriteLine("Current Tree: ");
    ...
}
```

3. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
4. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
5. Verify that the output in the console window resembles the following code example.

```
Current Tree:
1
3
4
4
5
7
Add 15
Current Tree:
1
3
4
4
5
7
15
Remove 5
Current Tree:
1
3
4
4
7
15
```

6. Press ENTER to close the console window.
7. Close Visual Studio:
   - In Visual Studio, on the **File** menu, click **Exit**.