- Describe the problem generics address.
  2. How would you create a list of strings, using the generic List class?
  3. How many generic type parameters does the Dictionary class have?
  4. True/False. When a generic class has multiple type parameters, they must all match.
  5. What method is used to add items to a List object?
  6. Name two methods that cause items to be removed from a List.
  7. How do you indicate that a class has a generic type parameter?
  8. True/False. Generic classes can only have one generic type parameter.
  9. True/False. Generic type constraints limit what can be used for the generic type.
  10. True/False. Constraints let you use the methods of the thing you are constraining to

- The problem generics address is that sometimes you want a class or method to be able to work with multiple data types, but without generics you would have to write separate versions of the code for each type. Generics allow you to create classes and methods that can work with different types, without sacrificing type safety.

- List<string> stringList = new List<string>();

- The Dictionary class has two generic type parameters: one for the type of the keys, and one for the type of the values.

- True. All generic type parameters in a generic class must be specified and must match when the class is instantiated.

- The Add method is used to add items to a List object

- Two methods that can cause items to be removed from a List are Remove and

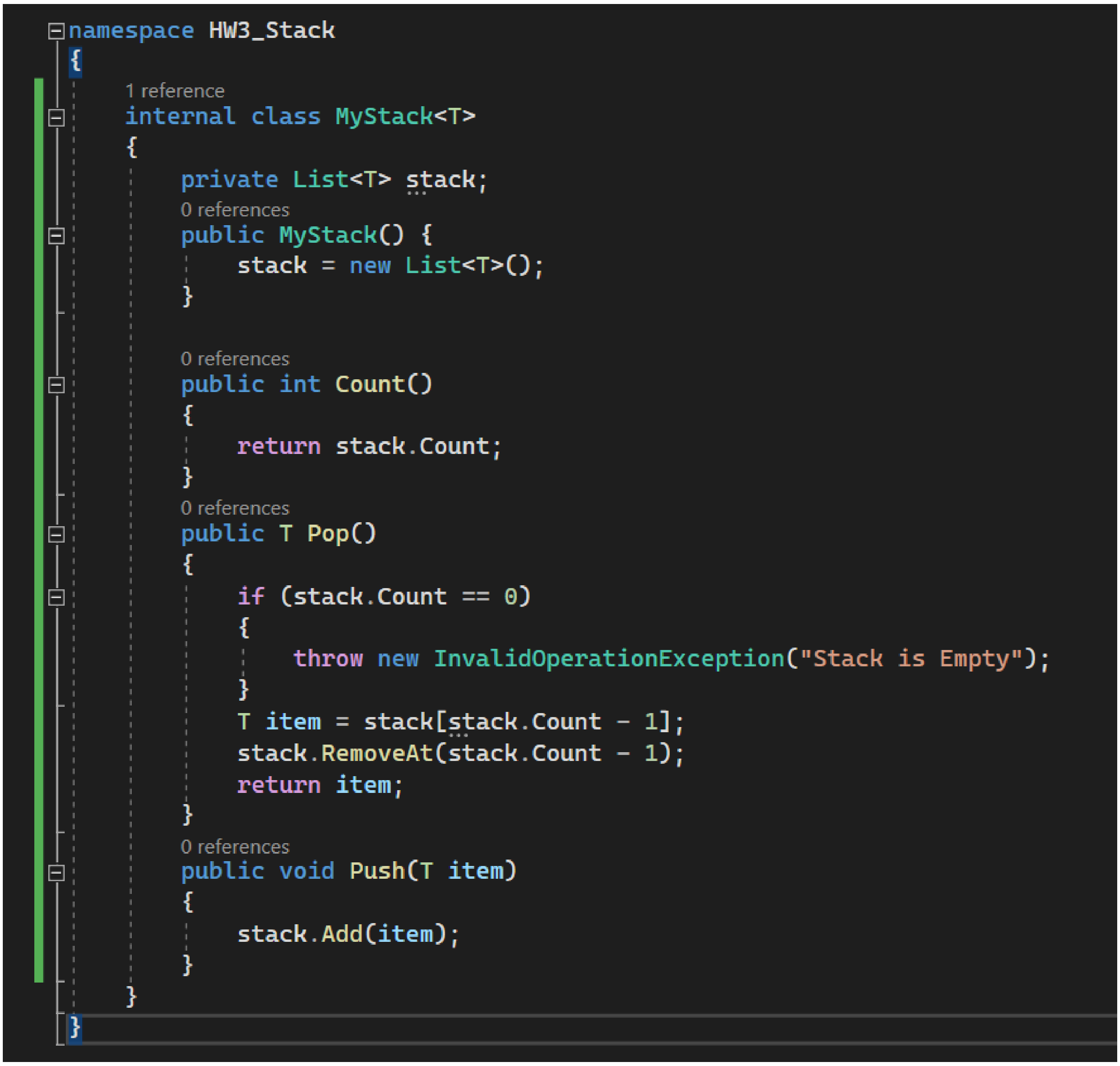
RemoveAt. Remove is used to remove a specific item from the list, while RemoveAt is used to remove an item at a specific index

- To indicate that a class has a generic type parameter, you use angle brackets < > after the class name, followed by the type parameter name

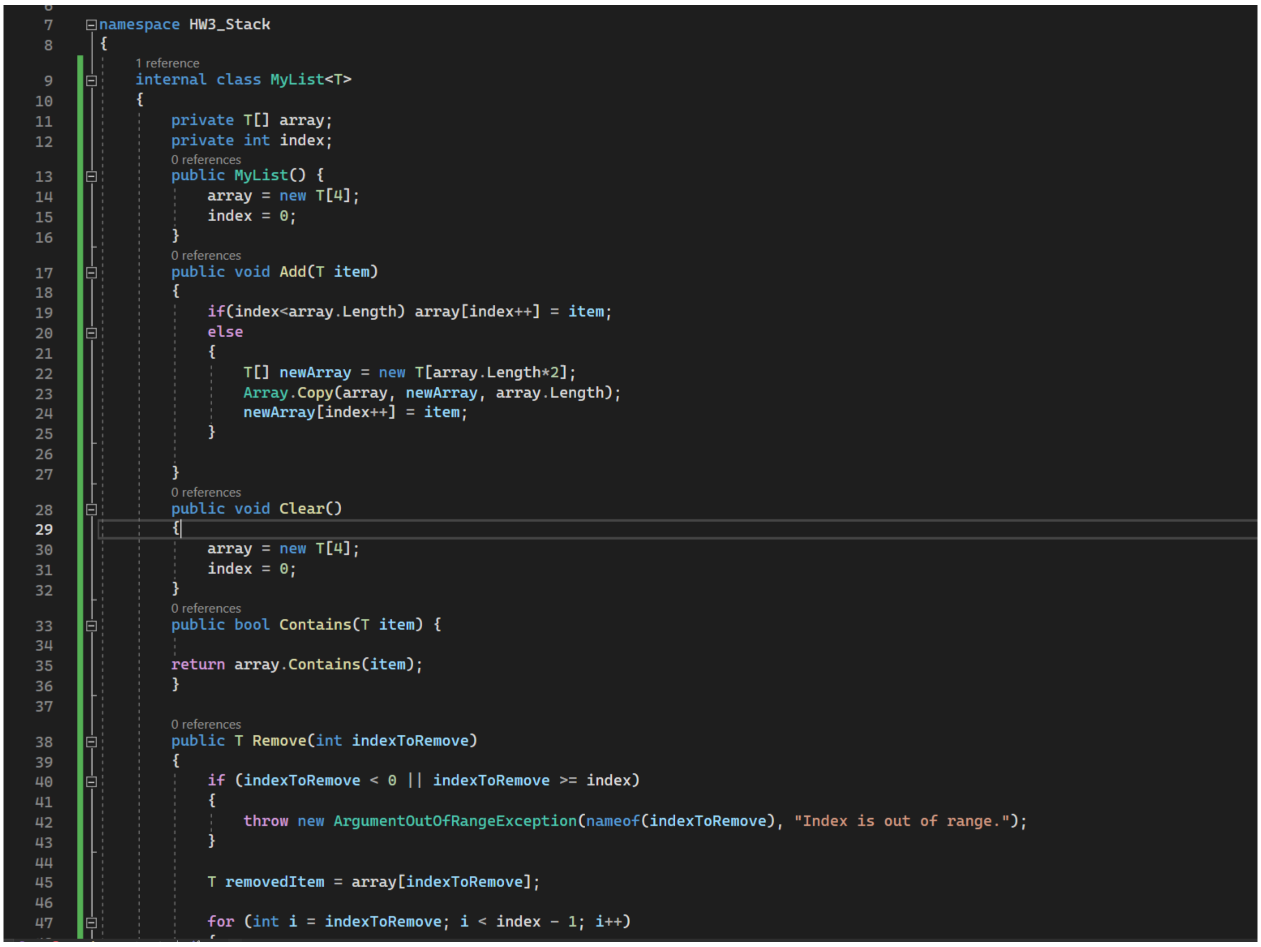
- False. Generic classes can have multiple generic type parameters, as long as each parameter is specified when the class is instantiated.

- True. Generic type constraints allow you to specify what types can be used for the generic type parameter. For example, you can specify that the type parameter must implement a specific interface, or that it must be a subclass of a specific class.

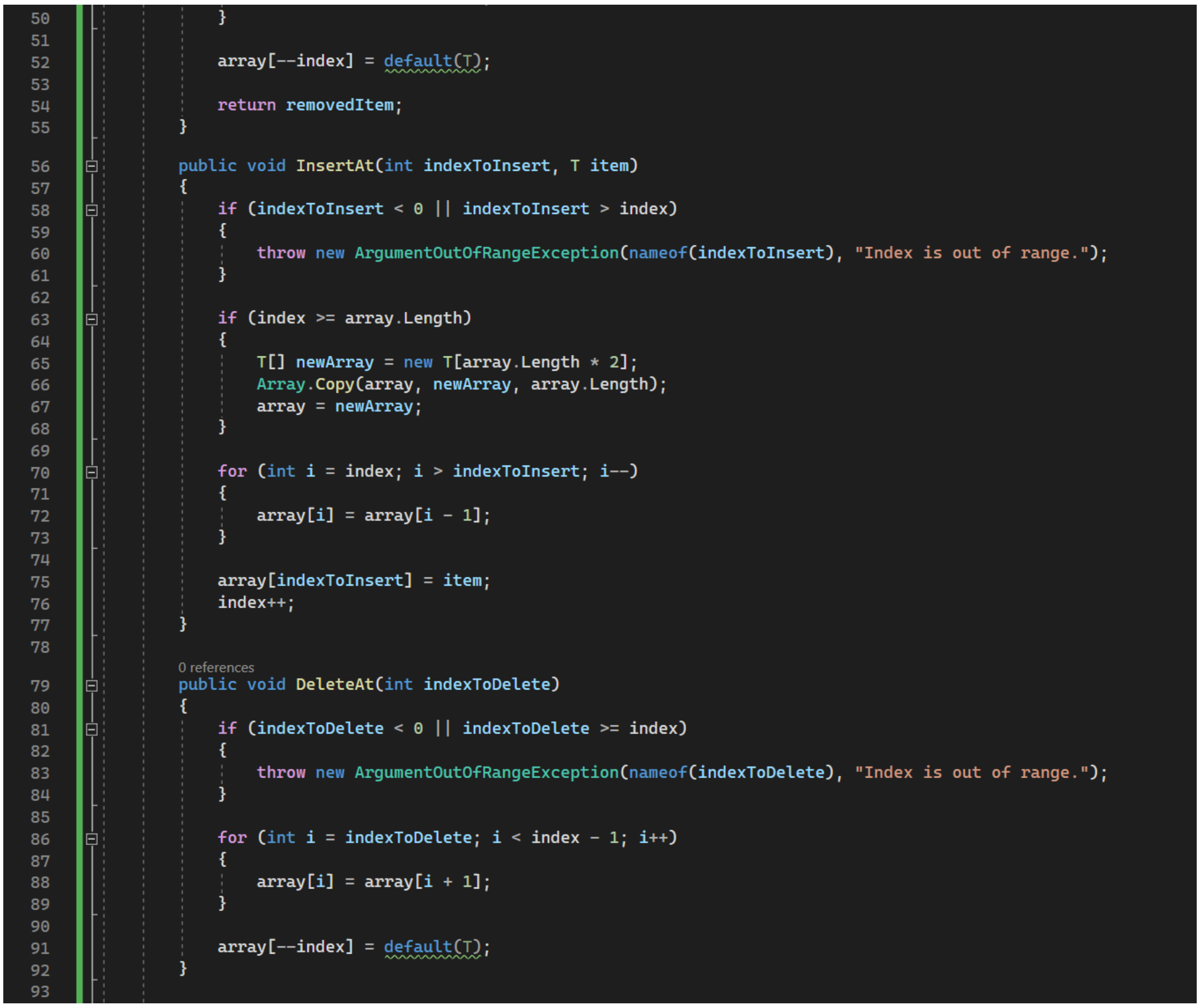
- True. Constraints let you use the methods of the thing you are constraining to. For example, if you constrain a generic type parameter to a class that has a specific method, you can call that method on instances of the class that are passed to the generic class or method.



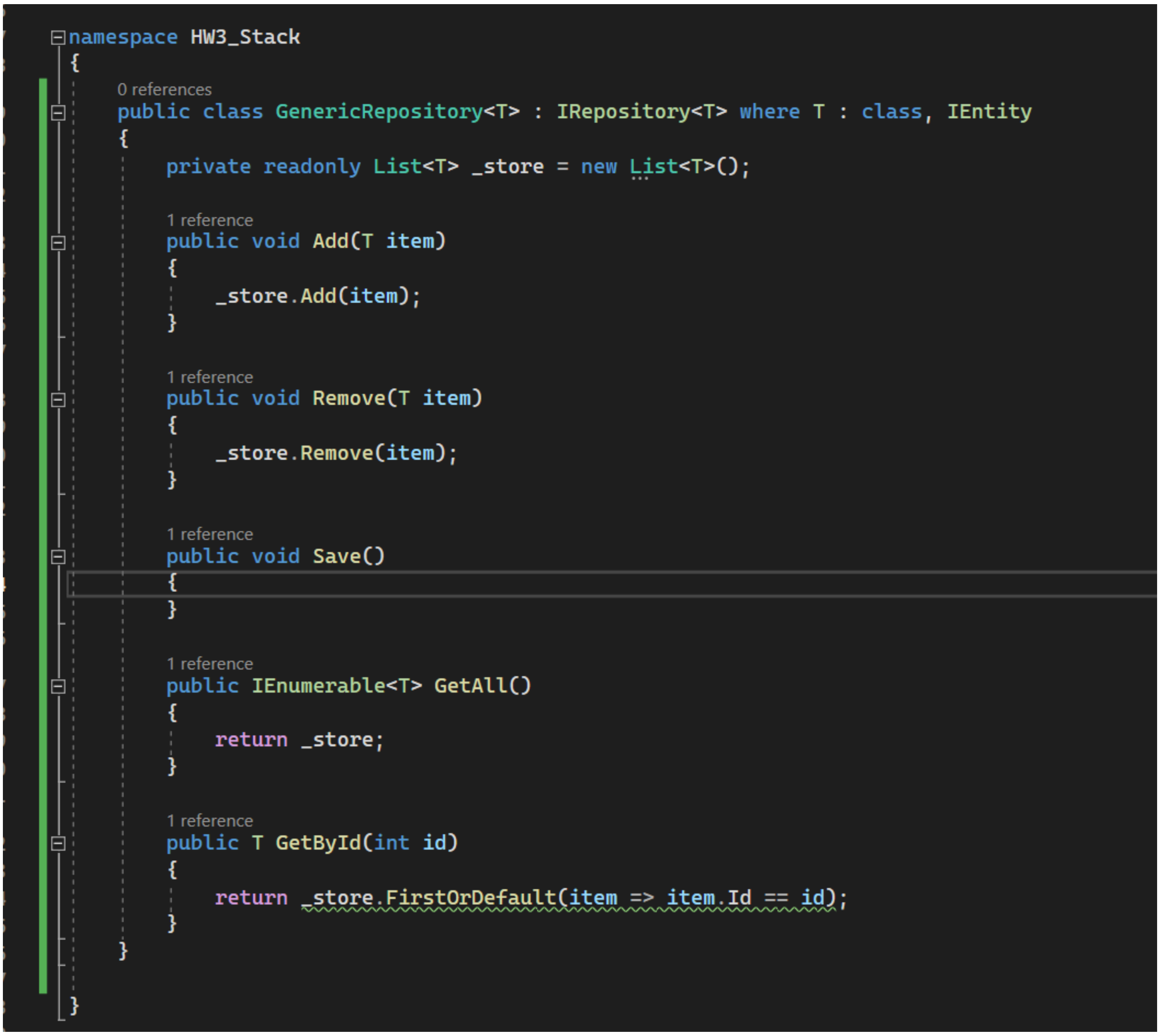
```
namespace HW3_Stack
{
    1 reference
    internal class MyStack<T>
    {
        private List<T> stack;
        0 references
        public MyStack() {
            stack = new List<T>();
        }

        0 references
        public int Count()
        {
            return stack.Count;
        }
        0 references
        public T Pop()
        {
            if (stack.Count == 0)
            {
                throw new InvalidOperationException("Stack is Empty");
            }
            T item = stack[stack.Count - 1];
            stack.RemoveAt(stack.Count - 1);
            return item;
        }
        0 references
        public void Push(T item)
        {
            stack.Add(item);
        }
    }
}
```



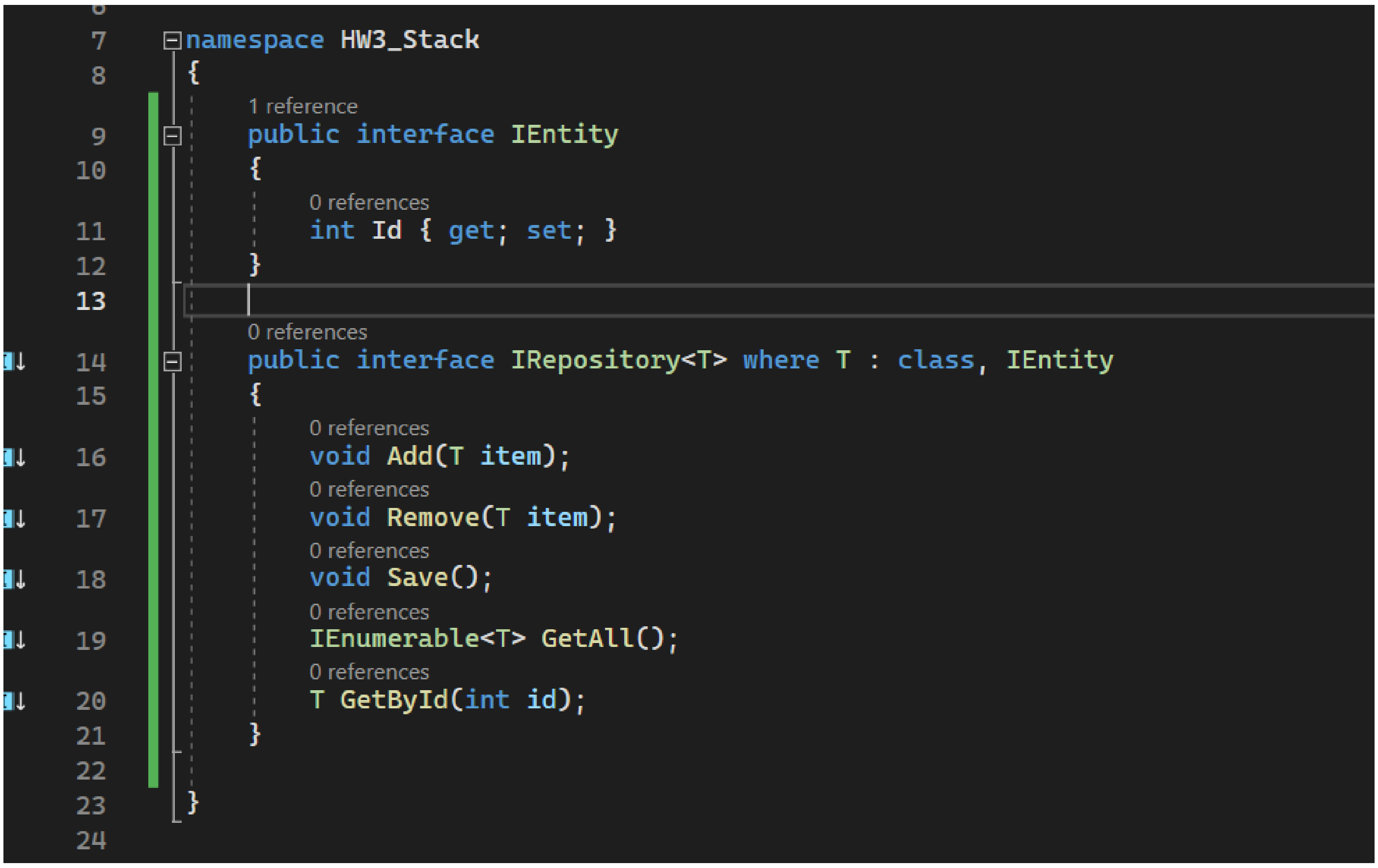
```
namespace HW3_Stack
{
    1 reference
    internal class MyList<T>
    {
        private T[] array;
        private int index;
        0 references
        public MyList() {
            array = new T[4];
            index = 0;
        }
        0 references
        public void Add(T item)
        {
            if(index<array.Length) array[index++] = item;
            else
            {
                T[] newArray = new T[array.Length*2];
                Array.Copy(array, newArray, array.Length);
                newArray[index++] = item;
            }

        }
        0 references
        public void Clear()
        {
            array = new T[4];
            index = 0;
        }
        0 references
        public bool Contains(T item) {

        return array.Contains(item);
        }

        0 references
        public T Remove(int indexToRemove)
        {
            if (indexToRemove < 0 || indexToRemove >= index)
            {
                throw new ArgumentOutOfRangeException(nameof(indexToRemove), "Index is out of range.");
            }

            T removedItem = array[indexToRemove];

            for (int i = indexToRemove; i < index - 1; i++)
```



```
            }

            array[--index] = default(T);

            return removedItem;
        }

        public void InsertAt(int indexToInsert, T item)
        {
            if (indexToInsert < 0 || indexToInsert > index)
            {
                throw new ArgumentOutOfRangeException(nameof(indexToInsert), "Index is out of range.");
            }

            if (index >= array.Length)
            {
                T[] newArray = new T[array.Length * 2];
                Array.Copy(array, newArray, array.Length);
                array = newArray;
            }

            for (int i = index; i > indexToInsert; i--)
            {
                array[i] = array[i - 1];
            }

            array[indexToInsert] = item;
            index++;
        }

        0 references
        public void DeleteAt(int indexToDelete)
        {
            if (indexToDelete < 0 || indexToDelete >= index)
            {
                throw new ArgumentOutOfRangeException(nameof(indexToDelete), "Index is out of range.");
            }

            for (int i = indexToDelete; i < index - 1; i++)
            {
                array[i] = array[i + 1];
            }

            array[--index] = default(T);
        }

```



```
namespace HW3_Stack
{
    0 references
    public class GenericRepository<T> : IRepository<T> where T : class, IEntity
    {
        private readonly List<T> _store = new List<T>();

        1 reference
        public void Add(T item)
        {
            _store.Add(item);
        }

        1 reference
        public void Remove(T item)
        {
            _store.Remove(item);
        }

        1 reference
        public void Save()
        {
        }

        1 reference
        public IEnumerable<T> GetAll()
        {
            return _store;
        }

        1 reference
        public T GetById(int id)
        {
            return _store.FirstOrDefault(item => item.Id == id);
        }
    }

}
```



```
namespace HW3_Stack
{
    1 reference
    public interface IEntity
    {
        0 references
        int Id { get; set; }
    }

    0 references
    public interface IRepository<T> where T : class, IEntity
    {
        0 references
        void Add(T item);
        0 references
        void Remove(T item);
        0 references
        void Save();
        0 references
        IEnumerable<T> GetAll();
        0 references
        T GetById(int id);
    }

}
```