1. Generics address the problem of type safety. They allow developers to write type-safe codes that do not commit to specific data types.
2. using System.Collections.Generic

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        List<string> stringList = new List<string>();

        stringList.Add("Hello");
        stringList.Add("World");
        stringList.Add("Hello"); // Adding another "Hello" to show the effect of Remove

        // Remove the first occurrence of "Hello"
        stringList.Remove("Hello");

        foreach (string str in stringList)
        {
            Console.WriteLine(str);
        }
    }
}
```

```
World
Hello

=== Code Execution Successful ===
```

3. Two generic types for KeyValuePair, i,e., Dictionary<TKey, TValue>
4. False. They don't have to be of the same type.
5. public void Add(T item).
6. public bool Remove(T item); public void RemoveAt(int index).
7. We use letter T surrounded by angle brackets, e.g., MyClass<T>.
8. False.
9. True. e.g., where T: struct limits T to be a struct.
10. True. The function can be called if it is guaranteed to exist due to the constraint.

1.

```
using System;
```

```
public class MyStack<T>
{
	private T[] items;
	private int top;

	public MyStack(int capacity)
	{
		items = new T[capacity];
		top = -1;
	}

	public int Count()
	{
		return top + 1;
	}

	public T Pop()
	{
		if (top == -1)
		{
			throw new InvalidOperationException(“Stack is empty.”);
		}
		T item = items[top];
		top--;
		return item;
	}

	public void Push(T item)
	{
		if (top == items.Length - 1)
		{
			throw new InvalidOperationException(“Stack overflow.”);
		}
		top++;
		items[top] = item;
	}
}

public struct MyStruct
{
	public int IntValue { get; set; }
	public bool BoolValue { get; set; }
	public string StringValue { get; set; }
```

```
    public override string ToString()
    {
        return $"[{IntValue}, {BoolValue}, {StringValue}]";
    }
}

class Program
{
    static void Main()
    {
        // Creating a MyStack of MyStruct
        MyStack<MyStruct> stack = new MyStack<MyStruct>(5);

        // Pushing items onto the stack
        for (int i = 0; i < 5; i++)
        {
            MyStruct item = new MyStruct
            {
                IntValue = i,
                BoolValue = i % 2 == 0, // Alternate between true and false
                StringValue = $"String {i}"
            };
            stack.Push(item);
            Console.WriteLine($"Pushed: {item}");
        }

        // Counting items in the stack
        Console.WriteLine($"Count: {stack.Count()}");

        // Popping items from the stack
        Console.WriteLine("Popping items:");
        while (stack.Count() > 0)
        {
            MyStruct item = stack.Pop();
            Console.WriteLine($"Popped: {item}");
        }
    }
}
```

```
Pushed: [0, True, String 0]
Pushed: [1, False, String 1]
Pushed: [2, True, String 2]
Pushed: [3, False, String 3]
Pushed: [4, True, String 4]
Count: 5
Popping items:
Popped: [4, True, String 4]
Popped: [3, False, String 3]
Popped: [2, True, String 2]
Popped: [1, False, String 1]
Popped: [0, True, String 0]

=== Code Execution Successful ===
```

2.

```
using System;

public class MyList<T>
{
    private T[] items;
    private int count;

    public MyList()
    {
        items = new T[10]; // Initial capacity
        count = 0;
    }

    public void Add(T element)
    {
        EnsureCapacity();
        items[count++] = element;
    }

    public T Remove(int index)
    {
        if (index < 0 || index >= count)
        {
            throw new IndexOutOfRangeException("Index is out of range.");
        }

        T removedItem = items[index];
        for (int i = index; i < count - 1; i++)
        {
            items[i] = items[i + 1];
        }
```

```
        count--;
        return removedItem;
    }

    public bool Contains(T element)
    {
        for (int i = 0; i < count; i++)
        {
            if (items[i].Equals(element))
            {
                return true;
            }
        }
        return false;
    }

    public void Clear()
    {
        Array.Clear(items, 0, count);
        count = 0;
    }

    public void InsertAt(T element, int index)
    {
        if (index < 0 || index > count)
        {
            throw new IndexOutOfRangeException("Index is out of range.");
        }

        EnsureCapacity();
        for (int i = count; i > index; i--)
        {
            items[i] = items[i - 1];
        }
        items[index] = element;
        count++;
    }

    public void DeleteAt(int index)
    {
        Remove(index);
    }

    public T Find(int index)
```

```
    {
        if (index < 0 || index >= count)
        {
            throw new IndexOutOfRangeException("Index is out of range.");
        }

        return items[index];
    }

    public int Count()
    {
        return count;
    }

    private void EnsureCapacity()
    {
        if (count == items.Length)
        {
            Array.Resize(ref items, items.Length * 2);
        }
    }
}

class Program
{
    static void Main()
    {
        MyList<double> myList = new MyList<double>();

        // Adding elements
        myList.Add(1.1);
        myList.Add(2.2);
        myList.Add(3.3);
        myList.Add(4.4);

        // Printing elements
        Console.WriteLine("Elements:");
        for (int i = 0; i < myList.Count(); i++)
        {
            Console.WriteLine(myList.Find(i));
        }

        // Removing element at index 1
        Console.WriteLine("\nRemoving element at index 1:");
```

```
            myList.Remove(1);

            // Printing elements after removal
            Console.WriteLine("Elements:");
            for (int i = 0; i < myList.Count(); i++)
            {
                Console.WriteLine(myList.Find(i));
            }

            // Checking if list contains 3.3
            Console.WriteLine("\nContains 3.3: " + myList.Contains(3.3));

            // Clearing the list
            Console.WriteLine("\nClearing the list...");
            myList.Clear();

            // Printing elements after clearing
            Console.WriteLine("Elements:");
            for (int i = 0; i < myList.Count(); i++)
            {
                Console.WriteLine(myList.Find(i));
            }
        }
}
```

```
Elements:
1.1
2.2
3.3
4.4

Removing element at index 1:
Elements:
1.1
3.3
4.4

Contains 3.3: True

Clearing the list...
Elements:

=== Code Execution Successful ===
```

3.

```
using System;
using System.Collections.Generic;
using System.IO;
```

```
public interface IRepository<T>
{
	void Add(T item);
	void Remove(T item);
	void Save();
	IEnumerable<T> GetAll();
	T GetById(int id);
}

public class GenericRepository<T> : IRepository<T>
{
	private Dictionary<int, T> items;
	private int nextId;

	public GenericRepository()
	{
		items = new Dictionary<int, T>();
		nextId = 1;
	}

	public void Add(T item)
	{
		items.Add(nextId, item);
		nextId++;
	}

	public void Remove(T item)
	{
		// Find the key associated with the item and remove it
		foreach (var pair in items)
		{
			if (EqualityComparer<T>.Default.Equals(pair.Value, item))
			{
				items.Remove(pair.Key);
				break;
			}
		}
	}

	public void Save()
	{
		string directoryPath = "/content/data";
		string filePath = "/content/data/items.txt";
```

```
            // Check if the directory exists, and create it if it doesn't
            if (!Directory.Exists(directoryPath))
            {
                Directory.CreateDirectory(directoryPath);
            }

            // Check if the file exists, and create it if it doesn't
            if (!File.Exists(filePath))
            {
                File.Create(filePath).Close(); // Close the file stream immediately after creating it
            }

            // Write the ID-item pairs to the file
            using (StreamWriter writer = new StreamWriter(filePath))
            {
                foreach (var pair in items)
                {
                    writer.WriteLine($"{pair.Key},{pair.Value}");
                }
            }

            Console.WriteLine("Changes saved to file.");
        }

        public IEnumerable<T> GetAll()
        {
            return items.Values;
        }

        public T GetById(int id)
        {
            if (items.ContainsKey(id))
            {
                return items[id];
            }
            else
            {
                throw new KeyNotFoundException($"Item with ID {id} not found.");
            }
        }
    }
```

```
class Program
{
    static void Main()
    {
        GenericRepository<string> repository = new GenericRepository<string>();

        // Adding items
        repository.Add("Item 1");
        repository.Add("Item 2");
        repository.Add("Item 3");

        // Removing an item
        repository.Remove("Item 2");

        // Saving changes
        repository.Save();

        // Getting all items
        IEnumerable<string> allItems = repository.GetAll();
        Console.WriteLine("All Items:");
        foreach (string item in allItems)
        {
            Console.WriteLine(item);
        }

        // Getting item by ID
        Console.WriteLine("Item with ID 1:");
        Console.WriteLine(repository.GetById(1));
    }
}
```

```
Changes saved.
All Items:
Item 1
Item 3
Item with ID 1:
Item 1

=== Code Execution Successful ===
```