

# Generics

Adding Type Safety and Code Reusability



**SoftUni Team**

**Technical Trainers**



**SoftUni**



**Software University**

<https://about.softuni.bg/>

[sli.do](https://sli.do)

**#csharp-advanced**

# Table of Contents

1. Generics
2. Generic Classes
3. Generic Methods
4. Generic Constraints






# Generics

Definition, Type Parameters and Safety

# What Are Generics?

- 
- Generics introduce the concept of **Type Parameters**
  - Allow designing classes and methods without **parameter type specification**
  - A generic **class** or a **method** accepts a certain type when it is **instantiated** by client code

```
public class CustomStack<T> { }  
CustomStack<int> =  
    new CustomStack<int>();
```

- Add **type safety** for the client
- Provide a powerful way to **reuse** code

```
List<int> strings = new List<int>();  
List<Person> people = new List<Person>();
```

- Example: we need a collection that will store only strings

```
List<string> strings = new List<string>();  
strings.Add(3); // Compile time error
```

- Blueprint for a **type** - **T (Type Parameter)**
- You can use it **anywhere** inside the **generic** class

```
class List<T>
{
    public Add (T element) {...}

    public T Remove () {...}

    public T Peek { get; }
}
```



# Generic Classes



```
public class ObjectList
{
    private object[] elements;
    public ObjectList ()
    { this.elements = new object[4]; }
    public void Add(object value){}
    public object Get(int index)
    {
        return this.elements[index];
    }
}
```

```
var objectList = new ObjectList();  
objectList.Add(1);  
objectList.Add(new Customer());  
objectList.Add(new Account());
```

```
var firItem = objectList[0]; // firItem is object  
var secItem = (Customer)objectList[1]; // cast
```

- Encapsulate operations to a **non-particular data type**
- Defined with **Type Parameters - T**

```
class List<T> { }  
class Stack<T> { }
```

- Most commonly used are **generic collections**:
  - Linked Lists, Hash tables, Stacks, Queues, Trees, etc.
  - Collections with **multiple** type **parameters** – Dictionary<T, V>



# Generic Methods

- Take a **certain** input and a **certain** output **type**

```
public class CustomerList
{
    public Customer Remove(Customer customer)
    {
        return removedCustomer;
    }
}
```

- Take **generic input** and return **generic output**

```
public List<T> CreateList<T>(T item)
{
    List<T> list = new List<T>();
    ...
    return list;
}
```

# Problem: Box of T

- Create a collection, that can store anything and has the following **methods**:
  - **Add()** should add on top of its contents
  - **Remove()** should remove the topmost element and **return it**
- It should have two public methods:
  - **void Add(T element)**
  - **T Remove()**
  - **int Count**

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1474#0>

# Solution: Box of T

```
public class Box<T>
{
    // TODO: Add fields and constructor
    public int Count => this.data.Count;

    public void Add(T item) { this.data.Add(item); }

    public T Remove() {
        var rem = this.data.Last();
        this.data.RemoveAt(this.data.Count - 1);
        return rem; }
}
```



# Problem: Generic Array Creator

- Create a class **ArrayCreator** with a single method:

```
static T[] Create(int length, T item)
```

- It should return an array with the given length
- Every element should be **set to the default item**

Check your solution here: <https://judge.softuni.org/Contests/Practice/Index/1474#1>

# Solution: Generic Array Creator

```
public static class ArrayCreator
{
    public static T[] Create<T>(int length, T item)
    {
        T[] array = new T[length];
        for(int i = 0; i < length; i++)
        { array[i] = item; }
        return array;
    }
}
```



# Generic Constraints

Apply Restrictions

- Constraints are represented in generics using **where**
- Restricting generic classes to **reference types** only:

```
public void MyMethod<T>()  
    where T : class
```

- **class** is the keyword

```
public void MyMethod<T>()  
    where T : struct
```

- **struct** is the keyword

- IL generated for **Equals<string>** would be different to that of **Equals<int>**

```
public static bool Equals<T> (T t1, T t2)
{
    return (t1 == t2);
}
```

- The case could be different if the **types** that are being compared have a **new definition of == operator**

- Specifying a **constructor** as a constraint

```
public void MyMethod<T>()  
    where T : new()
```

- Only a **default constructor** can be used in the constraints
- **Parameterized constructor** will be a **compilation error**

- Specifying a static **base class** as a constraint

```
public void MyMethod<T>()  
    where T : BaseClass  
{  
    ...  
}
```

- The type **argument** must **be or derive from** the **specified base class**

- Specifying **a generic base class** as a constraint

```
public void AddAll<TItem>(List<TItem> items)
    where TItem : T
{
    ...
}
```

- The **type argument** supplied for **T** must **be** or **derive from** the **argument** supplied for **TItem**
- **T** comes from the **generic class**



- Specifying **a generic base class** as a constraint

```
public void MyMethod<T>()  
    where T : BaseClass, new()  
{  
    ...  
}
```

- Invalid combination of constraints: **class** and **struct**

# Problem: Equality Scale

- Create a class **EqualityScale<T>** that:
  - Holds two elements: **left** and **right**
  - Receives the elements through its single constructor:
    - **EqualityScale(T left, T right)**
  - Has a method: **bool AreEqual()**
- The greater of the two elements is the heavier



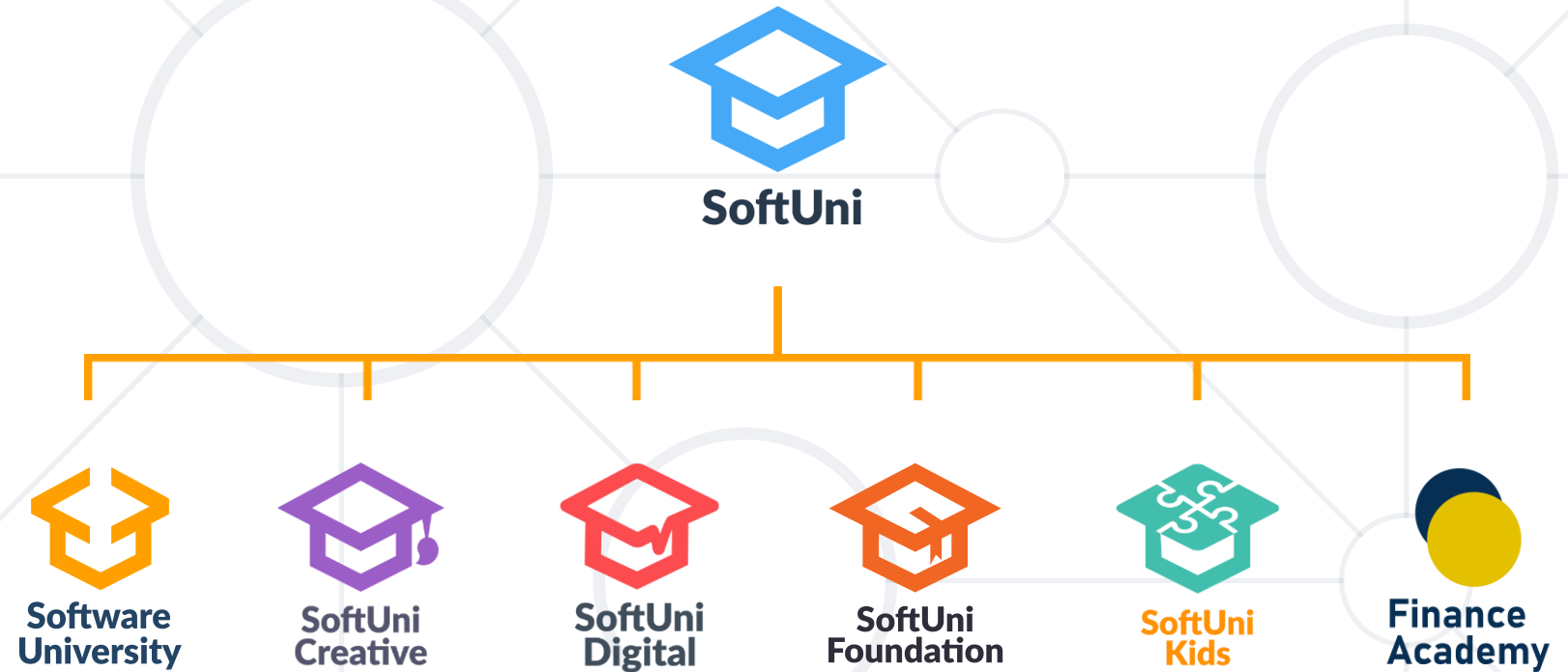
# Solution: Equality Scale

```
public class EqualityScale<T>
{
    private T left;
    private T right;
    public EqualityScale(T left, T right) {
        this.left = left;
        this.right = right;
    }
    public bool AreEqual() {
        bool result = this.left.Equals(this.right);
        return result;
    }
}
```

- **Generics** add type safety
- Generic code is more **reusable**
- Classes, interfaces and methods can be generic
- Generic **Constraints** can validate generic types



# Questions?



# SoftUni Diamond Partners



- Software University – High-Quality Education, Profession and Job for Software Developers

- [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)

- Software University Foundation

- [softuni.foundation](http://softuni.foundation)

- Software University @ Facebook

- [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)

- Software University Forums

- [forum.softuni.bg](http://forum.softuni.bg)



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>

