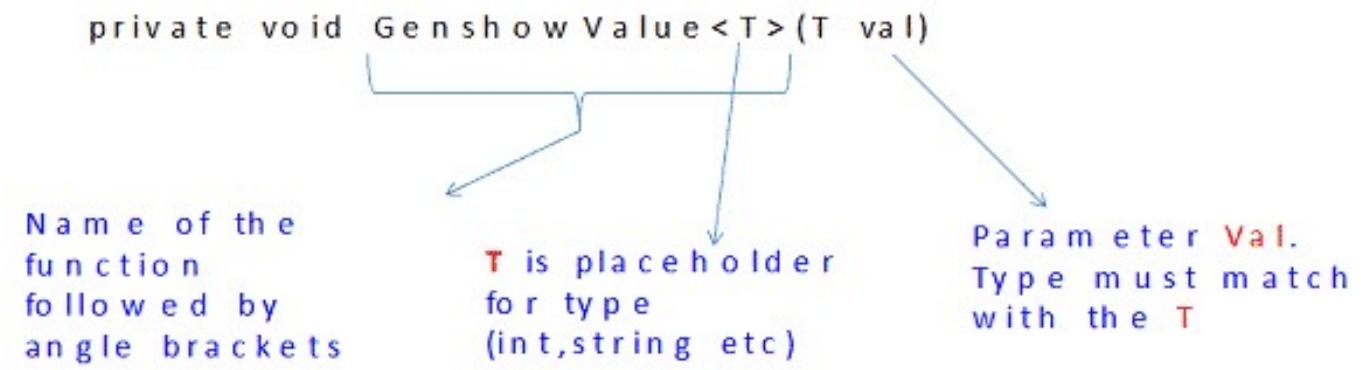
GENETIC PROGRAMMING .NET

GENERIC

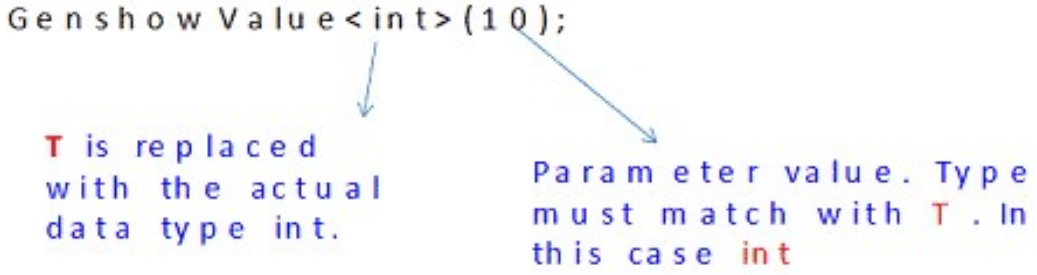
- Generic Collections:
  - Namespace: ***System.Collections.Generic***
  - *Stack<T>*
  - *Queue<T>*
  - *LinkedList<T>*
  - *SortedList<T>*
  - *List<T>*
  - *Dictionary<TKey, Tvalue>*

GENERIC

- The Generic Method:



The above method invoked as



GENERIC

- The Generic Method:

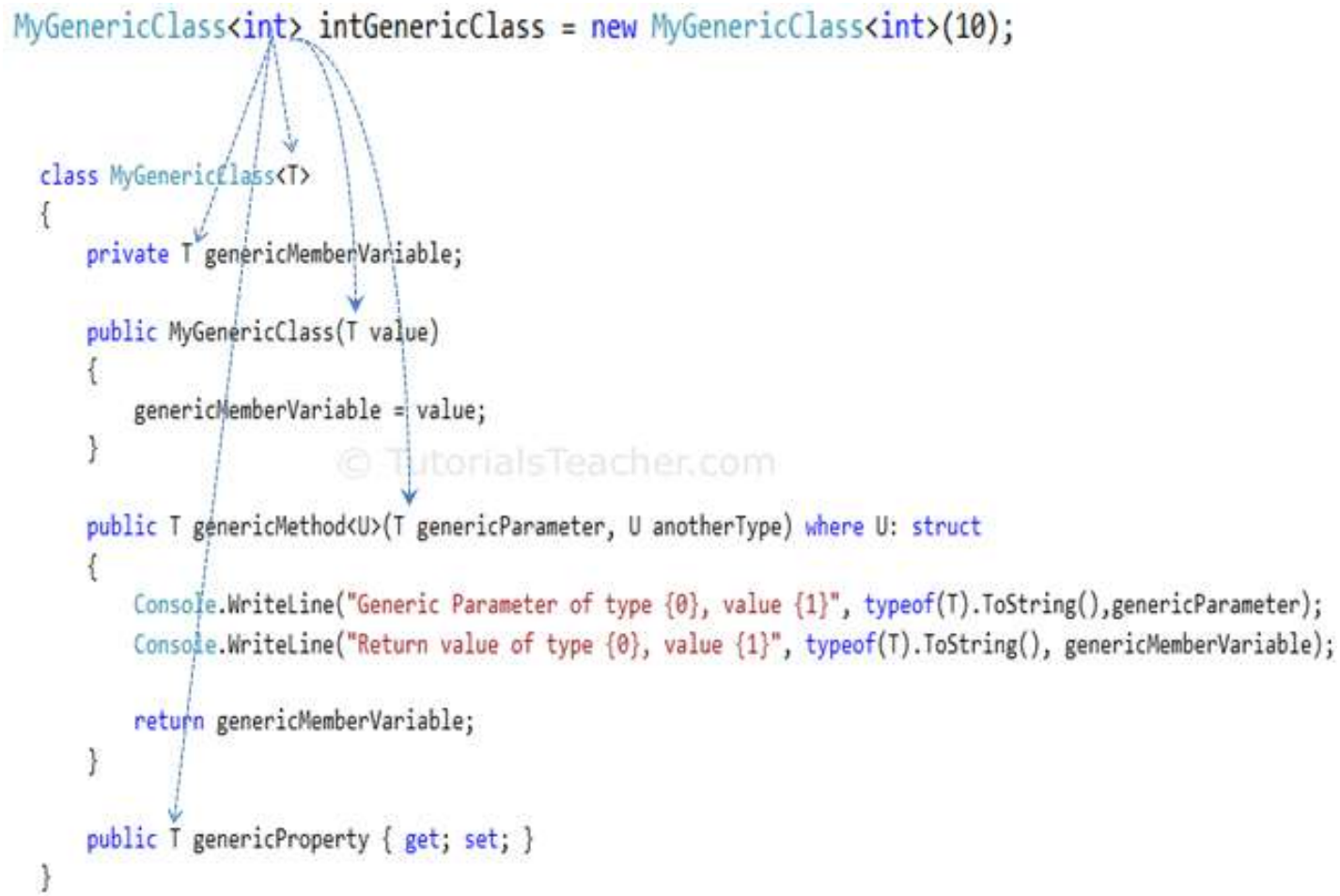
```
class Program
{
    static void Main(string[] args)
    {
        int a = 10, b = 90;
        Console.WriteLine("Before swap: {0}, {1}", a, b);
        Swap<int>(ref a, ref b);
        Console.WriteLine("After swap: {0}, {1}", a, b);

        // Swap 2 strings.
        string s1 = "Hello", s2 = "There";
        Console.WriteLine("Before swap: {0} {1}!", s1, s2);
        Swap<string>(ref s1, ref s2);
        Console.WriteLine("After swap: {0} {1}!", s1, s2);
        Console.ReadLine();

    }
    // This method will swap any two items.
    // as specified by the type parameter <T>.
    static void Swap<T>(ref T a, ref T b)
    {
        T temp;
        temp = a;
        a = b;
        b = temp;
    }
}
```

GENERIC

- The Generic Class:



```
MyGenericClass<int> intGenericClass = new MyGenericClass<int>(10);

class MyGenericClass<T>
{
    private T genericMemberVariable;

    public MyGenericClass(T value)
    {
        genericMemberVariable = value;
    }

    public T genericMethod<U>(T genericParameter, U anotherType) where U: struct
    {
        Console.WriteLine("Generic Parameter of type {0}, value {1}", typeof(T).ToString(),genericParameter);
        Console.WriteLine("Return value of type {0}, value {1}", typeof(T).ToString(), genericMemberVariable);

        return genericMemberVariable;
    }

    public T genericProperty { get; set; }
}
```

GENERIC

- **Default** value in Generic:

```
namespace SimpleGenerics
{
    // A generic Point structure.
    public struct Point<T>
    {
        // Generic state date.
        private T xPos;
        private T yPos;

        public Point(T xVal, T yVal)
        {
            xPos = xVal;
            yPos = yVal;
        }

        // Generic properties.
        public T X
        {
            get { return xPos; }
            set { xPos = value; }
        }

        public T Y
        {
            get { return yPos; }
            set { yPos = value; }
        }
```

```
        public override string ToString()
        {
            return string.Format("[{0}, {1}]", xPos, yPos);
        }

        // The 'default' keyword is overloaded in C# 2.0.
        // when used with generics, it represents the default
        // value of a generic parameter.
        public void ResetPoint()
        {
            xPos = 0;
            yPos = 0;
        }
    }
}
```

Error

```
        public override string ToString()
        {
            return string.Format("[{0}, {1}]", xPos, yPos);
        }

        // The 'default' keyword is overloaded in C# 2.0.
        // when used with generics, it represents the default
        // value of a generic parameter.
        public void ResetPoint()
        {
            xPos = default(T);
            yPos = default(T);
        }
    }
}
```

OK

GENERIC

- Constraints for Generic Type:

```
namespace CustomGenericCollection
{
    public class Car
    {
        // Constant for maximum speed.
        public const int maxSpeed = 100;

        // Internal state data.
        private int currSpeed;
        private string petName;

        Properties

        // Is the car still operational?
        private bool carIsDead;

        // A car has-a radio.
        private Radio theMusicBox = new Radio();

        public Car() { }
        public Car(string name, int currSp)
        {
            currSpeed = currSp;
            petName = name;
        }
```

```
namespace CustomGenericCollection
{
    public class CarCollection<T> : IEnumerable<T> where T : Car
    {
        private List<T> arCars = new List<T>();

        public T GetCar(int pos)
        { return arCars[pos]; }

        public void AddCar(T c)
        { arCars.Add(c); }

        public void ClearCars()
        { arCars.Clear(); }

        public int Count
        { get { return arCars.Count; } }

        public void PrintPetName(int pos)
        {
            Console.WriteLine(arCars[pos].PetName);
        }

        IEnumerable<T> / IEnumerable Members
    }
}
```

OK

GENERIC

- Constraints for Generic Type:

Generic Constraint	Description
where T : struct	The type parameter <T>must have System.ValueType in its chain of inheritance.
where T : class	<T>must be a reference type.
where T : new()	The type parameter <T>must have a default constructor and must be the last parameter.
where T : NameOfBaseClass	The type parameter <T>must be derived from the class specified by NameOfBaseClass.
where T : NameOfInterface	The type parameter <T>must implement the interface specified by NameOfInterface.

GENERIC

- Constraints Example:

```
// Contained items must have a default constructor.
public class MyGenericClass<T> where T : new()
{}

// Contained items must be a class implementing IDrawable
// and support a default ctor.
public class MyGenericClass<T> where T : class, IDrawable, new()
{}

// MyGenericClass derives from MyBase and implements ISomeInterface,
// while the contained items must be structures.
public class MyGenericClass<T> : MyBase, ISomeInterface where T : struct
{}

// <K> must have a default ctor, while <T> must
// implement the generic IComparable interface.
public class MyGenericClass<K, T> where K : new() where T : IComparable<T>
{}
```

Must be the last

GENERIC

- The Lack of Operator Constraints:

```
public class BasicMath<T>
{
    public T Add(T arg1, T arg2)
    { return arg1 + arg2; }

    public T Subtract(T arg1, T arg2)
    { return arg1 - arg2; }

    public T Multiply(T arg1, T arg2)
    { return arg1 * arg2; }

    public T Divide(T arg1, T arg2)
    { return arg1 / arg2; }
}
```

*A compiler error will apply any C# operators (+, -, *, ==, etc.) on the type parameters*

GENERIC

- The Lack of Operator Constraints:

```
public class BasicMath<T>
{
    public T Add(T arg1, T arg2)
    { return arg1 + arg2; }

    public T Subtract(T arg1, T arg2)
    { return arg1 - arg2; }

    public T Multiply(T arg1, T arg2)
    { return arg1 * arg2; }

    public T Divide(T arg1, T arg2)
    { return arg1 / arg2; }
}
```

*A compiler error will apply any C# operators (+, -, *, ==, etc.) on the type parameters*

Thank You !

IT FACULTY