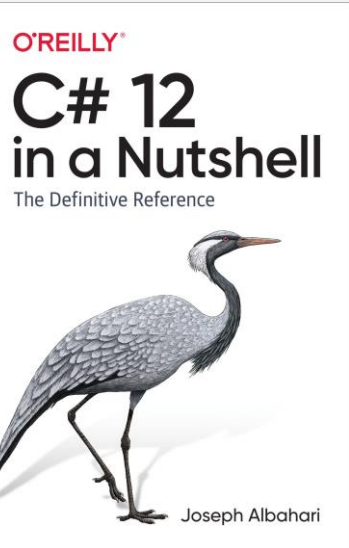


# Chapter 3 - Generics

- **Generics**
- **Generic Types**
- **Why Generics Exist?**
- **Generic Methods**
- **Declaring Type Parameters**
- **typeof and Unbound Generic Types**
- **The default Generic Value**
- **Generic Constraints**
- **Subclassing Generic Types**
- **Static Data**
- **Type Parameters and Conversions**
- **Covariance**



# Generics

C# has two separate mechanisms for writing code that is reusable across different types: inheritance and generics.

سی شارپ برای نوشتن کدهایی که امکان استفاده مجدد دارند دو مکانیزم مختلف معرفی کرده: inheritance و generics

Whereas inheritance expresses reusability with a base type, generics express reusability with a "template" that contains "placeholder" types.

(نتونستم براش ترجمه مناسبی پیدا کنم)

Generics, when compared to inheritance, can increase type safety and reduce casting and boxing.

Genreric ها در مقایسه با مبحث ارث بری میتوانند type safety رو افزایش بدند و casting و boxing رو کاهش بدند.

# Generic Types

A generic type declares type parameters—placeholder types to be filled in by the consumer of the generic type, which supplies the type arguments.

(نتونستم براش ترجمه مناسبی پیدا کنم)

Here is a generic type `Stack<T>`, designed to stack instances of type `T`.

در مثال زیر یک جنریک به نام `stack` با یک نوع `T` تعریف شده

```
0 references | 0 changes | 0 authors, 0 changes
public class Stack<T>
{
    int position;
    T[] data = new T[100];
    0 references | 0 changes | 0 authors, 0 changes
    public void Push(T obj) => data[position++] = obj;
    0 references | 0 changes | 0 authors, 0 changes
    public T Pop() => data[--position];
}
```

```
var intStack = new Stack<int>();
intStack.Push(5);
intStack.Push(10);
int x = intStack.Pop();           // x is 10
int y = intStack.Pop();           // y is 5
Console.WriteLine($"{x}, {y}"); // 10, 5
```

# Generic Types

```
var intStack = new Stack<int>();  
intStack.Push(5);  
intStack.Push(10);  
int x = intStack.Pop();           // x is 10  
int y = intStack.Pop();           // y is 5  
Console.WriteLine($"{x}, {y}"); // 10, 5
```

حالا میتونیم از جنریکی که ساختیم اینجوری استفاده کنیم.  
`Stack<int>` نوع پارامتر `T` رو برابر `int` قرار میده و یک نوع

On the fly ایجاد میکنه (در runtime)

اگه بخواهیم یک رشته متنی داخل `stack<in>` که تعریف کردیم

`Push` کنیم با خطا مواجه میشیم. در اصل `stack<int>` یه همچین تعریفی داره:

```
public class ###  
{  
    int position;  
    int[] data = new int[100];  
    public void Push (int obj) => data[position++] = obj;  
    public int Pop()          => data[--position];  
}
```

از نظر فنی ما به `stack<T>` میتونیم بگیم open-type و به

`Stack<int>` میتونیم بگیم close-type. موقع runtime همه

نمونه هایی که از جنریک ها ساخته شدند closed هستند (با نوعی که بهش دادیم پر شدند) به همین علت کد زیر اشتباهه:

```
var stack = new Stack<T>();    // Illegal: What is T?
```

# Generic Types

البته اگر در یک کلاس یا متد از پارامتر نامشخص  $T$  استفاده کنیم اوکی هست و کد بدون مشکل اجرا میشه:

```
2 references | 0 changes | 0 authors, 0 changes
public class Stack<T>
{
    int position;
    T[] data = new T[100];
    0 references | 0 changes | 0 authors, 0 changes
    public void Push(T obj) => data[position++] = obj;
    0 references | 0 changes | 0 authors, 0 changes
    public T Pop() => data[--position];
    0 references | 0 changes | 0 authors, 0 changes
    public Stack<T> Clone() => new Stack<T>(); // Legal
}
```

# Why Generics Exist?

جنریک ها هستند تا ما بتوانیم کدهایی بنویسیم که با نوع های مختلف بتوانیم ازش چندین و چندبار استفاده کنیم. فرض کنید ما نیاز به یک **stack** داریم که یه سری عدد رو داخلش نگه داریم و چیزی به اسم جنریک هم نداریم. یه راه اینه که برای هر نوع یه کلاس مجزا بسازیم. **stringStack.intStack** و غیره. این روش به وضوح باعث خلق کدهای تکراری خواهد شد. یه روش دیگه اینه که کلاسی براش تعریف کنیم که با نوع **object** کار میکنه:

```
0 references | 0 changes | 0 authors, 0 changes
public class ObjectStack
{
    int position;
    object[] data = new object[10];
    0 references | 0 changes | 0 authors, 0 changes
    public void Push(object obj) => data[position++] = obj;
    0 references | 0 changes | 0 authors, 0 changes
    public object Pop() => data[--position];
}
```

با این حال این کلاس به خوبی **intStack** برای نوع عددی کار نمیکند و برای استفاده از اون نیاز به **boxing** و **downcasting** داریم که موقع کامپایل چک نمیشه و ممکنه موقع **runtime** تازه به خطا بخوریم:

```
// Suppose we just want to store integers here:
var stack = new Generics.ObjectStack();

stack.Push("s");           // Wrong type, but no error!
int i = (int)stack.Pop();   // Downcast - runtime error
```

اینجاست که متوجه میشیم چقدر جنریک ها میتونند کار ما رو راحت کنند.

# Generic Methods

A generic method declares type parameters within the signature of a method.

With generic methods, many fundamental algorithms can be implemented in a general-purpose way.

متدهای جنریک هم مثل کلاسهای جنریک میتونند بهمون کمک کنند تا بسیاری از الگوریتم های پایه ای رو برای نوع های داده ای مختلف پیاده سازی کنیم.

```
2 references | - changes | -authors, -changes
static void Swap<T>(ref T a, ref T b)
{
    T temp = a;
    a = b;
    b = temp;
}

x = 5;
y = 10;
Swap(ref x, ref y);
Swap<int>(ref x, ref y);
```

مثال زیر دوتا متغیر از هرنوعی رو میتونه بگیره و جابه جاشون کنه. همونطور که میبینید دو مدل میتونیم متد swap رو فراخوانی کنیم.

# Generic Methods

Within a generic type, a method is not classed as generic unless it introduces type parameters (with the angle bracket syntax). The Pop method in our generic stack merely uses the type's existing type parameter, T, and is not classed as a generic method.

یک متد در کلاس جنریک، تا زمانی که خودمون تعریف نکنیم یک متد جنریک محسوب نمیشه، مثلاً متد Pop در کلاس جنریک stack که در چند اسلاید قبل تعریف کردیم صرفاً با نوعی کار میکنه که کلاس میشناسه و در دسته متدهای جنریک دسته بندی نمیشه.

Methods and types are the only constructs that can introduce type parameters. Properties, indexers, events, fields, constructors, operators, and so on cannot declare type parameters, although they can partake in any type parameters already declared by their enclosing type.

خلاصه اینکه فقط کلاس ها و متدها میتونند جنریک باشند و بقیه چیزها مثل پراپرتی ها یا رویدادها یا indexerها نمیتونند مستقلاً جنریک باشند. البته که در یک کلاس جنریک میتونند با نوع پارامتری که کلاس میگیره کار کنند.



# Generic Methods

In our generic stack example, for instance, we could write an indexer that returns a generic item:

برای مثال توی کلاس `stack` که تعریف کردیم میتونیم یه `indexer` بنویسیم که یه آیتم جنریک رو برگردونه:

```
3 references | 0 changes | 0 authors, 0 changes
public class Stack<T>
{
    int position;
    T[] data = new T[100];
    2 references | 0 changes | 0 authors, 0 changes
    public void Push(T obj) => data[position++] = obj;
    2 references | 0 changes | 0 authors, 0 changes
    public T Pop() => data[--position];
    0 references | 0 changes | 0 authors, 0 changes
    public Stack<T> Clone() => new Stack<T>(); // Legal
    1 reference | 0 changes | 0 authors, 0 changes
    public T this[int index] => data[index];
}
```

# Declaring Type Parameters

Type parameter ها میتوانند در کلاسها، struct ها، ایتترفیسها، delegate ها و متدها تعریف بشند.

بقیه ساختارها مثل پراپرتی ها نمیتونند اونها رو تعریف کنند ولی میتوانند ازشون استفاده کنند. به عنوان مثال در struct جنریک زیر پراپرتی values از T داره استفاده میکنه.

```
public struct Nullable<T>
{
    public T Value { get; }
}
```

جنریک ها میتوانند چندتا پارامتر داشته باشند و برای استفاده از اون به شکل زیر میتونید عمل کنید:

```
class Dictionary<TKey, TValue> {...}
```

```
Dictionary<int, string> myDict1 = new Dictionary<int, string>();
var myDict2 = new Dictionary<int, string>();
```

# Declaring Type Parameters

```
class A      {}  
class A<T>   {}  
class A<T1,T2> {}
```

جنریک ها میتوانند در صورتی که تعداد پارامترهایشان متفاوت باشند **overload** شوند.  
به این مثال دقت کنید.

معمولا جنریک هایی که یک پارامتر دارند آن پارامتر را با حرف **T** مشخص میکنیم و برای جنریک هایی که بیشتر از یک پارامتر دارند پارامترها را با پیشوند **T** و یک توصیف بیشتر تعریف میکنیم، مثل دیکشنری:

```
class Dictionary<TKey, TValue> {...}
```

# typeof and Unbound Generic Types

همونطور که در اسلایدهای قبلی اشاره شد جنریک ها موقع runtime دیگه open-type نیستند. اونها موقع کامپایل closed میشوند. با این حال میشه با استفاده از typeof یک جنریک رو unbound کرد. Open-generic ها برای reflection کاربرد دارند. همینطور میشه از typeof برای جنریک های closed هم استفاده کرد.

```
Type a1 = typeof(A<>);    // Unbound type (notice no type arguments).
Type a2 = typeof(A<,>);   // Use commas to indicate multiple type args.
Type a3 = typeof(A<int, int>);

Console.WriteLine(a1.ToString()); // CSharp12Nutshell.Chapter03.Generics.A`1[T]
Console.WriteLine(a2.ToString()); // CSharp12Nutshell.Chapter03.Generics.A`2[T1,T2]
Console.WriteLine(a3.ToString()); // CSharp12Nutshell.Chapter03.Generics.A`2[System.Int32,System.Int32]
```

# The default Generic Value

با استفاده از کلمه کلیدی **default** همیشه مقدار پیش فرض پارامتر یک جنریک رو دریافت کرد.  
مقدار پیش فرض برای **reference-type** ها **null** هست و برای **value-type** ها عدد صفر هست.

```
2 references | - changes | -authors, -changes
static string Zap<T>(T[] array)
{
    for (int i = 0; i < array.Length; i++)
        array[i] = default;

    return string.Join(",", array);
}

Console.WriteLine(Zap(new string[5])); // ",,,"
Console.WriteLine(Zap(new int[5]));    // "0,0,0,0,0"
```

# Generic Constraints

در حالت معمول شما میتونید پارامتر یک جنریک رو با هر نوعی جایگزین کنید. اما اگه بخواهیم میتونیم برای پارامترهای جنریک محدودیت بذاریم تا یک سری نوع های خاص رو فقط قبول کنند.

این لیست محدودیت هایی که معمولا میشه برای یک جنریک گذاشت:

```
where T : base-class // Base-class constraint
where T : interface // Interface constraint
where T : class // Reference-type constraint
where T : class? // (See "Nullable Reference Types" in Chapter 4)
where T : struct // Value-type constraint (excludes Nullable types)
where T : unmanaged // Unmanaged constraint
where T : new() // Parameterless constructor constraint
where U : T // Naked type constraint
where T : notnull // Non-nullable value type, or (from C# 8)
// a non-nullable reference type
```

# Generic Constraints

```
5 references | 0 changes | 0 authors, 0 changes
class SomeClass
{
    1 reference | 0 changes | 0 authors, 0 changes
    public SomeClass(int a)
    {
    }
}

2 references | 0 changes | 0 authors, 0 changes
class SomeClass2 : SomeClass, Interface1
{
    0 references | 0 changes | 0 authors, 0 changes
    public SomeClass2() : base(2)
    {
    }
}

2 references | 0 changes | 0 authors, 0 changes
interface Interface1 { }

1 reference | 0 changes | 0 authors, 0 changes
class GenericClass<T, U> where T : SomeClass, Interface1
                                where U : new()
{
}
}
```

توی این مثال `GenericClass<T,U>` یک پارامتر `T` میگیره که باید حتما از نوع `someClass` یا مشتقات اون باشه و حتما `Interface1` رو پیاده سازی کرده باشه و پارامتر `U` هم باید یک کلاس دارای `Constructor` بدون پارامتر باشه.

```
_ = new GenericClass<SomeClass2, SomeClass2>();
_ = new GenericClass<SomeClass, SomeClass>();
_ = new GenericClass<SomeClass2, SomeClass>();
_ = new GenericClass<SomeClass, SomeClass>();
```

در مثالهای بالا فقط مورد اول درسته چون هر دو پارامتر همه شرایط رو دارند ولی در بقیه موارد حداقل یک شرط نقض شده.

# Generic Constraints

A base-class constraint specifies that the type parameter must subclass (or match) a particular class;

محدودیت **base-class** همون چیزی بود که تو اسلاید قبل دیدیم. می‌گیم یه پارامتر حتما از نوع یه کلاس خاص باشه.

an interface constraint specifies that the type parameter must implement that interface.

اینم می‌گه پارامتر حتما باید نوعی باشه که یه ایترفیس خاصی رو پیاده سازی کرده باشه.

فرض کنید می‌خواهیم یه متد جنریک **Max** تعریف کنیم که بین دو مقدار بزرگترین رو برگردونه. برای اینکار از **Comparable<T>** استفاده میکنیم که در فضای نام **system** قرار دارد.

```
public interface Comparable<T>    // Simplified version of interface
{
    int CompareTo (T other);
}
```



# Generic Constraints

CompareTo اگر this (همون آبجکت اصلی) بزرگتر از آبجکت دوم باشه عدد مثبت برمیگردونه. حالا با استفاده از این اینترفیس میایم و متد جنریک Max رو به این شکل تعریف میکنیم:

```
2 references | - changes | -authors, -changes
static T Max<T>(T a, T b) where T : IComparable<T>
{
    return a.CompareTo(b) > 0 ? a : b;
}

Console.WriteLine(Max("a", "b"));
Console.WriteLine(Max(7, 8));
```

متد Max حالا هر نوعی که اینترفیس IComparable<T> رو پیاده سازی کرده باشه قبول میکنه. (تقریباً اکثر نوع های درونی در سی شارپ مثل int یا string این اینترفیس رو پیاده سازی کردند.)

# Subclassing Generic Types

نوع های جنریک مثل نوع های معمولی میتونند ارث برده بشند و به عنوان subclass ازشون استفاده بشه. کلاس مشتق شده میتونه کلاس والدش رو به صورت open یا close استفاده کنه:

```
0 references | 0 changes | 0 authors, 0 changes
class SpecialStack<T> : Stack<T> { } //open
0 references | 0 changes | 0 authors, 0 changes
class IntStack<T> : Stack<int> { } //close
```

```
0 references | 0 changes | 0 authors, 0 changes
class KeyedList<T, TKey> : List<T> { }
```

همینطور موقع ارث بری میتونیم هر تعداد پارامتر جدید که بخواهیم به subclass اضافه کنیم.

# Static Data

داده های استاتیک برای هر **closed-type** یونیک هستند. این مثال رو ببینید:

5 references | 0 changes | 0 authors, 0 changes

```
class Bob<T>
{
    public static int Count;
}
```

همینطور که میبینید برای هر نوع به صورت مجزا تعداد نگه داشته میشه

```
Console.WriteLine(++Bob<int>.Count);    // 1
Console.WriteLine(++Bob<int>.Count);    // 2
Console.WriteLine(++Bob<string>.Count);  // 1
Console.WriteLine(++Bob<object>.Count);  // 1
Console.WriteLine(++Bob<string>.Count);  // 2
```

# Type Parameters and Conversions

برای تبدیل نوع پارامترها در نوع های جنریک راههای زیر وجود دارد:

- Numeric conversion
- Reference conversion
- Boxing/unboxing conversion
- Custom conversion (via operator overloading; see Chapter 4)


The decision as to which kind of conversion will take place happens at compile time, based on the known types of the operands. This creates an interesting scenario with generic type parameters, because the precise operand types are unknown at compile time. If this leads to ambiguity, the compiler generates an error.

(اینو نفهمیدم چی میگه دقیقا اگه کمک کنید ممنون میشم)

# Type Parameters and Conversions

این مثال و ببینید. بدون دانستن نوع دقیق  $T$  کامپایلر نمیتونه تشخیص بده این تبدیلی که نوشتید آیا کار میکنه یا نه. برای همین خطا میده که من نمیتونم این کاری رو که ازم میخوای انجام بدم.

```
0 references | - changes | -authors, -changes
✓ StringBuilder Foo<T>(T arg)
{
    if (arg is StringBuilder)
        return (StringBuilder)arg; // Will not compile
}
```

 **class** System.Text.**StringBuilder**  
Represents a mutable string of characters. This class cannot be inherited.  
[GitHub Examples and Documentation \(Alt+O\)](#)  
CS0030: Cannot convert type 'T' to 'System.Text.StringBuilder'

# Type Parameters and Conversions

خب حالا راهکار چیه؟ ساده ترین راهکار استفاده از عملگر **as** هستش، اگه یادتون باشه تو بخش قبلی گفته بودیم این عملگر اگر نتونه کارشو انجام بده خطا نمیده و **null** برمیگردونه.

```
0 references | - changes | - authors, - changes
StringBuilder Foo<T>(T arg)
{
    //if (arg is StringBuilder)
    //    return (StringBuilder)arg;    // Will not compile

    var sb = arg as StringBuilder;
    if (sb != null) return sb;

    if (arg is StringBuilder)
        return (StringBuilder)(object)arg;

    return null;
}
```

یه راه دیگه اینه که اول **T** رو به آبجکت تبدیل کنیم و بعد به نوع دلخواه **cast** کنیم. اینجا هم جلوی خطا رو موقع کامپایل میگیریم ولی ممکنه موقع **runtime** به خطا بخوریم.

# Covariance

فرض کنید **A** میتونه به **B** تبدیل بشه، حالا اگر **X<A>** به **X<B>** هم قابل تبدیل باشه اونوقت میگیم **X** یک نوع **covariant** داره. برای مثال اگر این عبارت درست کار کنه **IFoo<T>** دارای **covariant** هست.

```
IFoo<string> s = ...;  
IFoo<object> b = s;
```

Covariance and contravariance (or simply “variance”) are advanced concepts. The motivation behind introducing and enhancing variance in C# was to allow generic interface and generic types (in particular, those defined in .NET, such as **IEnumerable<T>**) to work more as you’d expect. You can benefit from this without understanding the details behind covariance and contravariance.

داره میگه بدون اینکه بدونیم این قضیه **covariant** چی هست از مزایاش استفاده کردیم چون خود دات نت یه سریاش رو تو خودش تعریف کرده مثل **IEnumerable**

```
IEnumerable<String> strings = new List<String>();  
IEnumerable<Object> objects = strings;
```

# Variance is not automatic

سی شارپ برای اینکه مطمئن باشه **type-safety** ش زیر سوال نمیره، پارامترهای جنریک ها به صورت خودکار **variant** نیستند. این مثال و ببینید:

```
3 references | 0 changes | 0 authors, 0 changes  
class Animal { }  
2 references | 0 changes | 0 authors, 0 changes  
class Bear : Animal { }  
0 references | 0 changes | 0 authors, 0 changes  
class Camel : Animal { }
```

حالا اگه بخوایم با استفاده از کلاس **stack** که خودمون ساختیم موضوع **Variant** رو چک کنیم موقع کامپایل به خطا میخوریم:

```
Generics.Stack<Bear> bears = new Generics.Stack<Bear>();  
Generics.Stack<Animal> animals = bears;
```

#endregion

(local variable) Generics.Stack<Bear> bears

CS0029: Cannot implicitly convert type 'CSharp12Nutshell.Chapter03.Generics.Stack<CSharp12Nutshell.Chapter03.Generics.Bear>' to 'CSharp12Nutshell.Chapter03.Generics.Stack<CSharp12Nutshell.Chapter03.Generics.Animal>'

Show potential fixes (Ctrl+.)



# Variance is not automatic

و این باعث میشه اجازه نده همچین کدی بنویسیم و به خطای runtime بخوریم:

```
animals.Push(new Camel()); // Trying to add Camel to bears
```

با این وجود نبود covariance میتونه جلوی reusability کد رو بگیره. فرض کنید میخوایم یه متد بنویسیم که یه سری از حیوانات رو بشوریم:

```
1 reference | 0 changes | 0 authors, 0 changes
class ZooCleaner
{
    1 reference | 0 changes | 0 authors, 0 changes
    public static void Wash(Stack<Animal> animals) { }
}
```

حالا اگه بخوایم اختصاصا خرس ها رو بشوریم موقع کامپایل به خطا میخوریم.

```
ZooCleaner.Wash(new Generics.Stack<Bear>());
```



Generics.Stack<Bear>.Stack()

CS1503: Argument 1: cannot convert from 'CSharp12Nutshell.Chapter03.Generics.Stack<CSharp12Nutshell.Chapter03.Generics.Bear>' to 'CSharp12Nutshell.Chapter03.Generics.Stack<CSharp12Nutshell.Chapter03.Generics.Animal>'

# Variance is not automatic

حالا اگه بیاییم و متد رو به این شکل بنویسیم میتونیم یه کد reusable داشته باشیم و بدون مشکل هر نوع حیوانی رو بشوریم:

1 reference | 0 changes | 0 authors, 0 changes

```
public static void Wash<T>(Stack<T> animals) where T : Animal { }
```

```
ZooCleaner.Wash(new Generics.Stack<Bear>());  
ZooCleaner.Wash(new Generics.Stack<Camel>());  
ZooCleaner.Wash(new Generics.Stack<Animal>());
```

# Arrays

بنا به دلایل تاریخی (historical گفته تو متن اصلی) آرایه ها از covariance پشتیبانی میکنند. یعنی B[] میتونه به A[] تبدیل بشه اگر B یک subclass از A باشه و البته هر دو reference-type باشند.

```
Bear[] bears = new Bear[3];
Animal[] animals = bears; // OK
animals[0] = new Camel(); // Runtime error ✖
```

#endregion

Exception Unhandled

**System.ArrayTypeMismatchException:** 'Attempted to access an element as a type incompatible with the array.'

[Show Call Stack](#) | [View Details](#) | [Copy Details](#) | [Start Live Share session](#)

▶ [Exception Settings](#)

با این وجود اگه بیایم یه خونه از آرایه رو که خرس توش هست بخوایم شتر بریزیم توش به خطای runtime برمیخوریم:

# Declaring a covariant type parameter

با استفاده از کلمه کلیدی **out** در پارامتر اینترفیس میتونیم قابلیت **covariant** رو به اون اضافه کنیم. حواسمون باشه که این موضوع برای کلاس های جنریک شدنیه نیست.

این کلمه کلیدی باعث میشه که **T** در موقعیت خروجی قرار بگیره

```
2 references | 0 changes | 0 authors, 0 changes
public interface IPoppable<out T> { T Pop(); }

10 references | 0 changes | 0 authors, 0 changes
public class Stack<T> : IPoppable<T>
{
    int position;
    T[] data = new T[100];
    3 references | 0 changes | 0 authors, 0 changes
    public void Push(T obj) => data[position++] = obj;
    4 references | 0 changes | 0 authors, 0 changes
    public T Pop() => data[--position];
    0 references | 0 changes | 0 authors, 0 changes
    public Stack<T> Clone() => new Stack<T>(); // Legal
    1 reference | 0 changes | 0 authors, 0 changes
    public T this[int index] => data[index];
}
```

```
var bears = new Generics.Stack<Bear>();
bears.Push(new Bear());
// Bears implements IPoppable<Bear>. We can convert to IPoppable<Animal>:
IPoppable<Animal> animals = bears; // Legal
Animal a = animals.Pop();
```