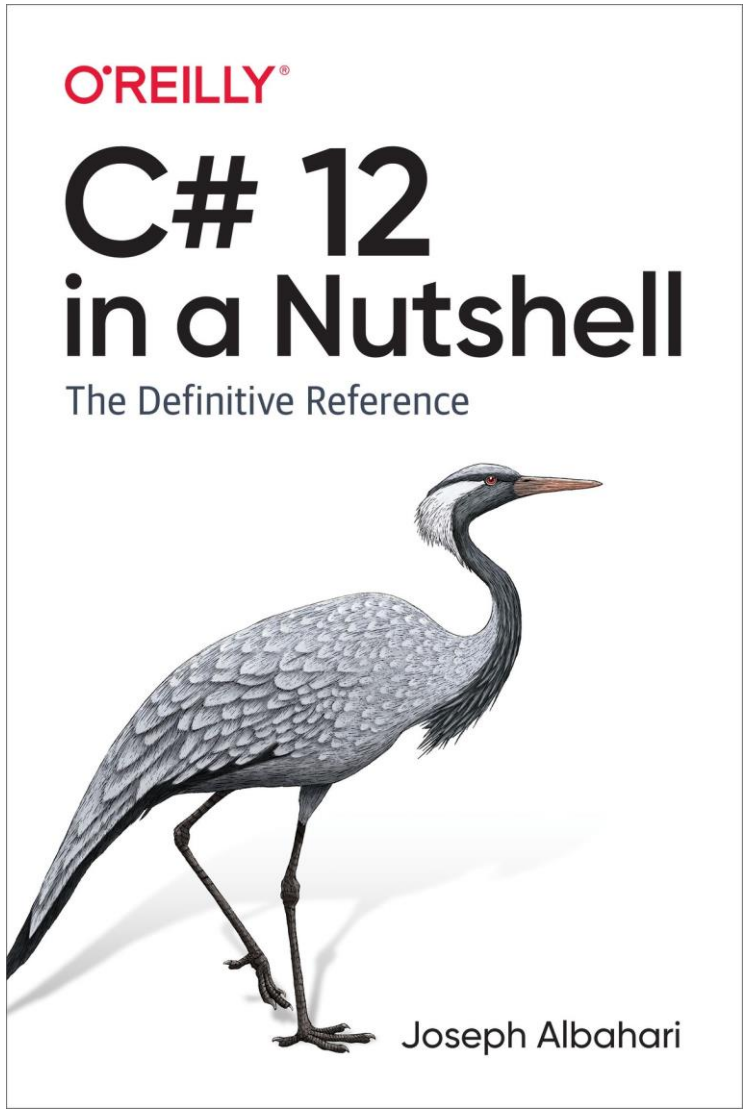
Chapter 4 (Patterns)

- Type Patterns
- Constant Pattern
- Relational Patterns
- Pattern Combinators
- var Pattern
- Tuple and Positional Patterns
- Property Patterns
- List Patterns



Type Patterns

قبلا دیدیم که میتونیم یه همچین الگویی داشته باشیم برای چک کردن نوع یک متغیر:

```
if (obj is string)
    Console.WriteLine(((string)obj).Length);

if (obj is string s)
    Console.WriteLine(s.Length);
```

به این الگو به اصطلاح میگن type-pattern.

کلمه کلیدی is میتونه الگوهای دیگه ای رو هم پشتیبانی کنه مثل Property-pattern که در نسخه های جدید C# معرفی شده:

```
if (obj is string { Length: 4})
    Console.WriteLine("A string with 4 characters");
```

الگوها در این سه حالت پشتیبانی میشن:

- After the is operator (variable is pattern)
- In switch statements
- In switch expressions

Constant Pattern

این الگو به شما اجازه میده آبجکت رو مستقیم با یک مقدار ثابت مقایسه کنید:

```
if (obj is 2)
    Console.WriteLine("obj is 2");

if (obj is int && (int)obj == 2)
    Console.WriteLine("obj is 2");
```

این مثال رو ببینید:

هر دو شرط یک کار رو دارند میکنند و عملا شرط دوم کاری رو انجام میده که شرط اول داره پشت صحنه خودش انجام میده.

طبیعیه که C# بهمون اجازه نده تا از عملگر == برای مقایسه آبجکت با یک مقدار ثابت استفاده کنیم چرا که کامپایلر نیاز داره تا نوع متغیر رو از قبل بدونه.

این الگو، یک جایگزین منطقی تر داره به این شکل:

```
if (2.Equals(obj))
    Console.WriteLine("obj is 2");
```

در ادامه خواهیم دید که این الگو وقتی با الگوهای دیگه ترکیب میشه، میتونه مفیدتر باشه.

Relational Patterns

از C# 9 به بعد ما میتونیم از عملگرهای <، >، =< و => در الگوهامون استفاده کنیم.

```
if (obj is > 100)
    Console.WriteLine("x is greater than 100");
```

این الگو به شکل معناداری در دستورات switch میتونه بهمون کمک کنه.

```
string Weight = obj switch
{
    < 18.5m => "underweight",
    < 25m => "normal",
    < 30m => "overweight",
    _ => "obese"
};

Console.WriteLine($"Weight: {Weight}");
```

همچنین این الگو هم مثل الگوی قبلی نوع متغیر براش مهمه، تو مثال زیر داخل obj مقدار decimal ریختیم پس نمیتونه با int مقایسه بشه و نتیجه به شکل زیر میشه:

```
obj = 2m;                       // obj is decimal
Console.WriteLine(obj is < 3m);  // True
Console.WriteLine(obj is < 3);   // False
```

در ادامه خواهیم دید که این الگو هم مثل الگوی قبل در ترکیب با سایر الگوها میتونه مثمر ثمرتر باشه.

Pattern Combinators

از C# 9 به بعد میتونیم از عملگرهای or، and و not برای ترکیب الگوها استفاده کنیم.

```
0 references | - changes | -authors, -changes
bool IsJanetOrJohn(string name) => name.ToUpper() is "JANET" or "JOHN";

0 references | - changes | -authors, -changes
bool IsVowel(char c) => c is 'a' or 'e' or 'i' or 'o' or 'u';

0 references | - changes | -authors, -changes
bool Between1And9(int n) => n is >= 1 and <= 9;

4 references | - changes | -authors, -changes
bool IsLetter(char c) => c is >= 'a' and <= 'z'
                         or >= 'A' and <= 'Z'
                         and not 'b' and not 'B';

Console.WriteLine(IsLetter('A')); // True
Console.WriteLine(IsLetter('B')); // False
Console.WriteLine(IsLetter('z')); // True
Console.WriteLine(IsLetter('2')); // False
```

مشابه عملگرهای && و || اینجا هم عملگر and تقدم بالاتری نسبت به عملگر or داره که میتونید با استفاده از پرانتز اونو تغییر بدید.

یه حرکت قشنگی که میشه با عملگر not زد اینه که ببینیم متغیر یک نوع خاص نباشه:

```
if (obj is not string)
    Console.WriteLine("obj is not string");
```

var Pattern

این الگو یک نوع type-pattern هست که به جای نوع متغیر از کلمه کلیدی var استفاده میکنه. در این الگو تبدیل همیشه موفقیت آمیز است، بنابراین هدفش اینه تا بتونیم از متغیر جدیدی که تولید میکنه استفاده مجدد کنیم:

```
0 references | - changes | -authors, -changes
bool IsJanetOrJoe(string name) =>
  name.ToUpper() is var upper && (upper == "JANET" || upper == "JOE");

//bool IsJanetOrJoe(string name)
//{
//    string upper = name.ToUpper();
//    return upper == "JANET" || upper == "JOHN";
//}
```

- ✓ در این مثال متد کامنت شده همان کار متد بالایی را انجام میدهد.

Tuple and Positional Patterns

این الگو در C# 8 معرفی شد و میتونه در switch هایی که برای چند مقدار نیاز داریم استفاده بشه:

```
3 references | - changes | -authors, -changes
int AverageCelsiusTemperature(Season season, bool daytime) =>
  (season, daytime) switch
  {
      (Season.Spring, true) => 20,
      (Season.Spring, false) => 16,
      (Season.Summer, true) => 27,
      (Season.Summer, false) => 22,
      (Season.Fall, true) => 18,
      (Season.Fall, false) => 12,
      (Season.Winter, true) => 10,
      (Season.Winter, false) => -2,
      _ => throw new Exception("Unexpected combination")
  };

Console.WriteLine(AverageCelsiusTemperature(Season.Summer, false)); //22
Console.WriteLine(AverageCelsiusTemperature(Season.Summer, true));  //27
Console.WriteLine(AverageCelsiusTemperature(Season.Winter, false)); //-2
```

Tuple and Positional Patterns

این الگو میتونه برای نوعهایی که متد Deconsruct رو دارند هم استفاده بشه و به خوبی خودشو باهاشون سازگار کنه:

```
1 reference | 0 changes | 0 authors, 0 changes
public class PointClass(int x, int y)
{
    1 reference | 0 changes | 0 authors, 0 changes
    public int X { get; } = x;
    1 reference | 0 changes | 0 authors, 0 changes
    public int Y { get; } = y;

    0 references | 0 changes | 0 authors, 0 changes
    public void Deconstruct(out int x, out int y)
    {
        x = X;
        y = Y;
    }
}

0 references | 0 changes | 0 authors, 0 changes
public record PointRecord(int x, int y); // Has compiler-generated deconstructor
```

توی این مثال ما یه کلاس داریم که متد Deconstruct رو خودمون براش نوشتیم و یک رکورد داریم که خودش خودکار اونو تولید میکنه و میبینید که در هر دو مورد به درستی کار میکنه:

```
var pointClass = new PointClass(2, 2);
var pointRecord = new PointRecord(2, 2);

Console.WriteLine(pointClass is (2, 2));  // True
Console.WriteLine(pointRecord is (2, 2));  // True
```

Tuple and Positional Patterns

حالا بیاییم یه مثال از ترکیب این الگو و type-pattern با استفاده از دستور switch بنویسیم:

```
5 references | - changes | -authors, -changes
string Print(object obj) => obj switch
{
    PointClass(0, 0) => "Empty point",
    PointClass(var x, var y) when x == y => "Diagonal",
    PointClass(var x, var y) when x > y => "X is great",
    PointClass(var x, var y) when x < y => "y is great",
    _ => "No Support"
};

Console.WriteLine(Print((2, 2)));                 // "No Support"
Console.WriteLine(Print(new PointClass(2, 2))); // "Diagonal"
Console.WriteLine(Print(new PointClass(2, 5))); // "X is great"
Console.WriteLine(Print((2M, 2M)));               // "No Support"
Console.WriteLine(Print(('A','C')));              // "No Support"
```

Property Patterns

این الگو میتونه براساس پراپرتی های یک آبجکت عمل کنه، یه نمونه ساده ازشو قبلا دیدیم:

```
if (obj is string { Length: 4})
    Console.WriteLine("A string with 4 characters");
```

این الگو با استفاده از دستور switch میتونه خیلی مفید باشه، این مثال رو ببینید:

```
0 references | - changes | -authors, -changes
bool ShouldAllow(Uri uri) => uri switch
{
    { Scheme: "http", Port: 80 } => true,
    { Scheme: "https", Port: 443 } => true,
    { Scheme: "ftp", Port: 21 } => true,
    { Scheme: { Length: 4 }, Port: 80 } => true,
    { Scheme.Length: 5, Port: 443 } => true,
    { Host: { Length: < 1000 }, Port: > 0 } => true,
    { Scheme: "http" } when string.IsNullOrWhiteSpace(uri.Query) => true,
    { IsLoopback: true } => true,
    _ => false
};
```

در این الگو شما میتونید تفزیبا هرکاری بکنید، این الگو رو با الگوهای دیگه ترکیب کنید، مقایسه های خاص انجام بدید و خلاصه اینکه دستتون برای انجام هرکاری بازه:

Property Patterns

همینطور میتونید این الگو رو در ترکیب با type-pattern به کار ببرید:

```
0 references | - changes | -authors, -changes
bool ShouldAllow(object uri) => uri switch
{
    Uri { Scheme: "http", Port: 80 } httpUri => httpUri.Host.Length < 1000,
    Uri { Scheme: "https", Port: 443 } httpUri when httpUri.Host.Length < 1000 => true,
    Uri { Scheme: "http", Port: 8080, Host: string host } => host.Length < 1000,
    Uri { Scheme: "http", Port: 8081, Host: var host } => host.Length < 1000,
};
```

List Patterns

این الگو با همه نوعهای collection که پراپرتی count یا length را دارند و indexable هستند میتونه کار کنه.

```
int[] numbers = { 0, 1, 7, 8, 4 };
Console.WriteLine(numbers is [0, 1, 7, 8, 4]);   // True
Console.WriteLine(numbers is [0, 1, _, _, 4]);   // True
Console.WriteLine(numbers is [0, 1, var x, 8, 4] && x > 1);   // True
Console.WriteLine(numbers is [0, .., 4]);    // True
Console.WriteLine(numbers is [0, .. var mid, 4] && mid.Contains(7)); // True
```

علامت _ به معنی هر مقداری است.

Var pattern اینجا هم کار میکنه و میتونه با این الگو ترکیب بشه.

دو نقطه یک slice ایجاد میکنه، یک slice میتونه با صفر یا بیشتر آیتم خودشو match کنه.

Slice رو به راحتی میتونید با var-pattern ترکیب کنید و شروط جدیدی خلق کنید.

لازم به ذکر هست که یک list-pattern فقط میتونه یک slice داشته باشه.