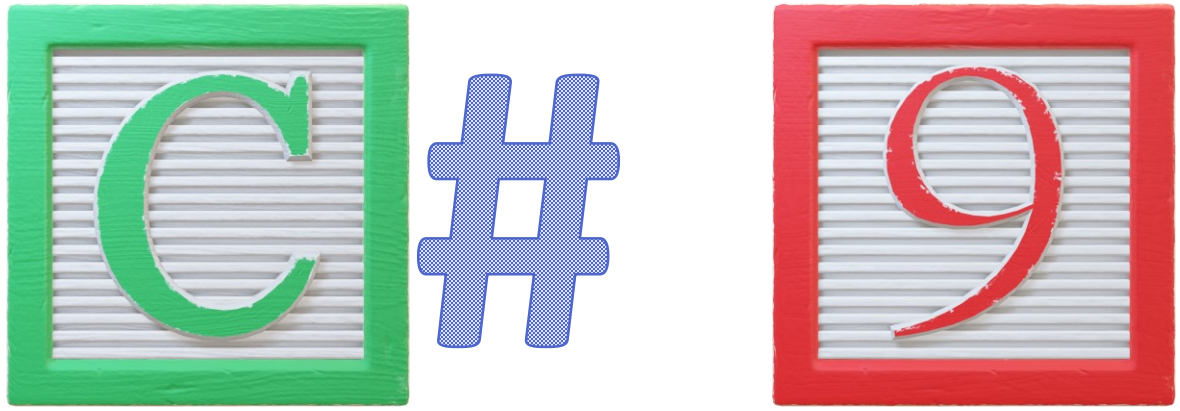

Cheat Sheet

https://github.com/alugili

28.11.2020

Records

A new C# type is immutable by default. The equality between Records is compared by structure or by reference.

```
// Default Record.
public record Person(string Name, int Age);

// Mutable Record.
public record Person(string Name, int Age)
{
    public string Name { get; set; } = Name;
    public int Age { get; set; } = Age;
}
var person1 = new Person("Bassam Alugili",42 );
var person2 = new Person("Bassam Alugili",42);

Console.WriteLine(person1 == person2); // True; Structural equality.
Console.WriteLine(person1.Equals(person2)); // True; Structural equality.

// False; Referential equality.
Console.WriteLine(ReferenceEquals(person1, person2));

var person3 = person1 with { Age = 43 }; // Change the default record! -->
Create a new one!
Console.WriteLine(person1 == person3); // False; Structural equality.

// Change the mutable record.
person1.Age = 43;
var (name, age) = person3; // Destruct a record.
```

Pros:
Records are a lightweight type that can remove a lot of code.
Structural equality and Referential equality.

Cons:
Allocating a lot of objects.

Pattern Matching Enhancements

C# 9 contains six main changes in pattern matching. Type patterns enhancements and all other patterns are introduced in C# 9.

Type patterns is used to match the input against a type. If the input type is a match to the type specified in the pattern, the match succeeds. C# 9 removes the type pattern followed by another pattern restriction.
Combinator Patterns Permit the programmer to combine multiple patterns on one line, with AND/OR operators, or to negate a pattern by using the NOT operator.
- **Conjunctive patterns** represent the logical “and” of the two sub-patterns pattern1 and pattern2.
- **Disjunctive patterns** represent the logical “or” of the two sub-patterns pattern1 and pattern2.
- **Negated “not” pattern** that requires a given pattern not to match.

Parenthesized patterns permit the programmer to put parentheses around any pattern.
Relational patterns permit the programmer to match their input against constant values to determine if the input is > or < or = to that constant.

Type Patterns

```
object checkType = new int();
var getType = checkType switch
{
 string => "string", // match against the type only.
 int => "int",
  _ =>   "obj"
};

Console.WriteLine(getType); // Output: int
```

Or patterns

```
var person = new Person("Bassam", 43);
var ageInRange = person switch
{
    Person(_, < 18) => "less than 18",
    (_, 18) or (_, > 18) => "18 or greater"
};
// Output: 18 or greater.
Console.WriteLine(ageInRange);
```

And patterns

```
var person = new Person("Bassam", 43);

var ageInRange = person switch
{
 Person(_, < 18) => "less than 18",
 ("Bassam",_) and (_,>18)=>"Bassam is greater than 18"
};

// Output: Bassam is greater than 18
Console.WriteLine(ageInRange);
```

Negated not patterns

```
var person = new Person("Bassam", 43);
var meOrNot = person switch
{
    not ("Bassam", 43) => "not me!",
    _=>"Me :-)"
};

Console.WriteLine(meOrNot);
```

Parenthesized patterns

```
public record IsNumber(bool Isvalid, int Number);
var is10 = new IsNumber(true, 11);
var n10 = is10 switch
{
  ((_,>1 and <5) and (_,>5 and <9)) or (_,10) => "10",
   _=> "not 10"
};
Console.WriteLine(n10); // Output: 10
```

Relational Patterns

```
var person = new Person("Bassam", 43);
var person2 = new Person("Thomas", 4);
var ageInRange = person switch
{
  Person(_, < 18) => "less than 18",
  (_, > 18) => "greater than 18",
  (_, 18) => "18 years old!"
};
// Output: greater than 18
Console.WriteLine(ageInRange);
```

Native Ints

C# 9 adds two new data types (*nint, nuint*). The new types are depending on the host platform and the compilation settings

```
nint nativeInt = 55;
Console.WriteLine(nint.MaxValue);

Compiled in x86 Configuration
Output: 2147483647

Compiled in x64 Configuration
Output: 9223372036854775807
```

Pros:
Make C# more compatible with Mac and iOS APIs.

Cons:
A lot of C# developers are not familiar with this concept.

Target New Types

This feature allows you to omit the type of the object you are instantiating.

```
Point p = new(1, 1);
ConcurrentDictionary<int, int> dict = new();
Point[] ps = {new(1, 1),new(2, 2),new(3, 3)};
```

Pros:
Less code.

Cons:
It can make your code harder to read.

Init Only Setters

This feature allows you to create an object in the nominal code style. Object initializer belongs to the nominal category.

```
public class InitDemo
{
   public string Start { get; init; }
   public string Stop { get; init; }
}
// Creating the object with the nominal style.
var initDemo = new InitDemo
{
    Start = "S1",
    Stop = "S2"
};
```

Pros:
Nice syntax and it removes some code overhead.

Target-typed Conditional Expressions

This feature allows you the implicit conversion from the null coalescing expression.

```
void M(int[] list, uint? u)
{
  int[] x = list ?? (int[]) new[] {1,2};
  var l = u ?? -1; // C# 9 in C# 8 you need -1u
}
```

Localsinit

In C# 9, you can use the new attribute *SkipLocalsInit* to instruct the compiler to suppress the emitting *.locals* init flag. This attribute applies at module, class, or method level.

```
[System.Runtime.CompilerServices.SkipLocalsInit]
static unsafe void DemoLocalsinit()
{
    int x;

    // Take care! x is not initialized!
    Console.WriteLine(*&x);
}
```

Pros:
Improves the performance of the methods.

Cons:
The impact on the method performance in most cases is small. Please use it only if you know exactly what you are doing! And with profiling.

GetEmurator

This feature allows you to create an extension method to allow foreach loops on *IEnumerator<T>* and *IAsyncEnumerator<T>* interfaces.

```
public static class Extensions
{
    public static IEnumerator<T> GetEnumerator<T>(this IEnumerator<T> enumerator) => enumerator;
}
```

With the above extension you can do:

```
 IEnumerator<string> enumerator = new Collection<string> { "Bassam", "Thomas", "Volker" }.GetEnumerator();

foreach (var guru in enumerator)
{
    Console.WriteLine($"Welcome {guru}!");
}
```

Lambda Discard Parameters

This feature allows you to add multiple discards (_,_) to be used as ignored parameters of lambdas and anonymous methods.

```
Func<int,int,int> zero = (_,_) => 0;
```

Pros:
Syntax Sugar.

Attributes on Local Functions

The idea is to permit attributes to be part of the declaration of a local function.

```
static void Main(string[] args)
{
   [Conditional("DEBUG")]
   // NotNull-attribute.
   static void DoSomething([NotNull] string test)
   {
     Console.WriteLine("Do it!");
   }

  DoSomething (“Doing!");
}
```

Extending Partial Methods

This feature allows you to remove the restrictions of the partial method (*void, out, accessibility*).

```
partial class Doing
{
 internal partial bool DoSomething(string s, out int i);
}

partial class Doing
{
 internal partial bool DoSomething(string s, out int i)
    {
      i = 0;
      return true;
    }
}
```

Static Anonymous Functions

This feature allows you to use the static keyword for lambdas to prevent capturing locals and parameters

```
public int number = 5;
….
Func<string> toString = () => number.ToString();
Console.WriteLine(toString());
```

The field number is captured by the anonymous lambda function and can cause an unintended allocation. To solve that you can add the modifier static on the lambda and use the const modifier on the field (number).

```
public const int number = 5;
….
Func<string> toString =static () => number.ToString();
Console.WriteLine(toString());
```

Pros:
Anonymous methods need allocations and this feature might help to make the anonymous methods more performance.

Module Initializers

The module initializer code is executed when an assembly is Loaded/ Initialized. You can compare it with the static constructor in C#, but in the case of module initializers, the method is executed only once for the entire assembly.

```
[ModuleInitializer]
public static void DoSomethingBeforeMain()
{
   Console.WriteLine("Huhu");
}
```

Covariant Return Types

Covariant return types is a feature in which a subclass of a type can be specified in a method override in a derived type; in other words, the overridden method has a more specific return type than the declaration in the base type.

```
public virtual Person GetPerson()
{
    // This is the parent (or base) class.
    return new Person();
}

public override Person GetPerson()
{
  //You can return the child class, but still return a Person
    return new Student();
}
```

Now, you can return the more specific type in C# 9.

```
public virtual Person GetPerson()
{
    // This is the parent (or base) class.
    return new Person();
}

public override Student GetPerson()
{
    // Better!
    return new Student();
}
```

Pros:
It can help you to remove a lot of ugly typecasting.

About me

Bassam Alugili

Senior Software Specialist/Database Expert at STRATEC SE
alugili@gmail.com
https://github.com/alugili
www.bassam.ml