Урок 8

Урок 8

Дополнительные возможности языка C#

Дополнительные возможности языка C#.

Введение

- Checked и unchecked
- Кортежи
- Переменные out и пустые переменные
- Локальные переменные и возвращаемые значения Ref
- Локальные функции
- Замыкание
- Паттерны switch
- Объявления using
- Span
- Lazy
- Индексы и диапазоны
- Ковариантность и контравариантность

Практическое задание

Дополнительные материалы

Используемая литература

На этом уроке:

1. Научимся использовать кортежи
2. Познакомитесь с фишками языка C#

Введение

На данном уроке мы разберем несколько полезных фишек языка C#.

Checked и unchecked

Строка кода C# может выполняться как в проверенном, так и в непроверенном контексте. В проверенном контексте арифметическое переполнение создает исключение. В непроверяемом контексте арифметическое переполнение игнорируется и результат усекается путем удаления старших разрядов, которые не помещаются в целевой тип данных.

- checked. Укажите проверенный контекст.
- unchecked. Укажите непроверенный контекст.

Следующие операции не затрагиваются проверкой переполнения.

- Выражения, использующие следующие предопределенные операторы на целочисленных типах: ++, --; унарные -, +, -, *, /.
- Явные числовые преобразования между целочисленными типами либо из float или double в целочисленный тип.

Если ни checked, ни unchecked не указаны, контекст по умолчанию для неконстантных выражений (выражения, вычисляемые во время выполнения) определяется по значению параметра компилятора -checked. По умолчанию значение этого параметра не задано, и арифметические операции выполняются в непроверенном контексте.

Для константных выражений (выражения, которые можно полностью вычислить во время компиляции) контекстом по умолчанию является проверяемый. Если только константное выражение явным образом не размещено в непроверенном контексте, переполнения, возникающие при вычислении выражения во время компиляции, вызывают ошибки времени компиляции.

```
using System;
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleChecked
    {
        public void Main()
        {
            int a = 150;
            int b = 150;

            try
            {
                byte result = checked((byte)(a + b));
            }
```

```
            catch (Exception e)
            {
                Debug.Log(e.Message);
            }
        }
    }
}
```

Кортежи

C# предоставляет расширенный синтаксис для классов и структур, который используется для объяснения цели проекта. Однако в некоторых случаях расширенный синтаксис требует дополнительной работы с минимальной результативностью. Зачастую требуется написание методов, которым нужна простая структура, состоящая из более чем одного элемента данных. Для поддержки этих сценариев в C# были добавлены кортежи. Кортежи — это упрощенные структуры данных, содержащие несколько полей для представления элементов данных. Поля не проверяются, и собственные методы определять нельзя.

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleTuple
    {
        private sealed class Player
        {
            private int _maxHP;
            private int _currentHP;

            public (int currentHP, int maxHP) GetHP()
            {
                return (_currentHP, _maxHP);
            }
        }

        public void Main()
        {
            (int currentHP, int maxHP) playerHp = (42, 100);

            Debug.Log($"{playerHp.currentHP}/{playerHp.maxHP}");
        }
    }
}
```

Метод GetHP класса Player возвращает кортеж из двух целых чисел. В методе Main класса ExampleTuple продемонстрирован функционал того как можно присвоить значение кортежу. Теперь немного перепишем

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleTuple
    {
        private sealed class Player
        {
            private int _maxHP;
            private int _currentHP;
```

```
            public Player()
            {
                _maxHP = 100;
                _currentHP = 42;
            }

            public (int currentHP, int maxHP) GetHP()
            {
                return (_currentHP, _maxHP);
            }
        }

        public void Main()
        {
            Player player = new Player();
            (int currentHP, int maxHP) playerHp = player.GetHP();

            Debug.Log($"{playerHp.currentHP}/{playerHp.maxHP}");
        }
    }
}
```

Еще есть класс **Tuple**, но он не удобен в плане того, что не именованы параметры

```
public void Main()
{
    var playerHp = new Tuple<int, int>(42, 100);

    Debug.Log($"{playerHp.Item1}/{playerHp.Item2}");
}
```

Метод Deconstruct предоставляет набор аргументов out для каждого из свойств, которые нужно извлечь.

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleTuple
    {
        private sealed class Player
        {
            private int _maxHP;
            private int _currentHP;

            public Player()
            {
                _maxHP = 100;
                _currentHP = 42;
            }

            public void Deconstruct(out int currentHP, out int maxHP) =>
                (currentHP, maxHP) = (_currentHP, _maxHP);
        }

        public void Main()
        {
            Player player = new Player();
            (int currentHP, int maxHP) = player;

            Debug.Log($"{currentHP}/{maxHP}");
```

```
        }
    }
}
```

Переменные out и пустые переменные

Вот так выглядит выпуск луча вперед из позиции курсора. Оговоримся что Camera.main лучше закешировать.

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleRaycast
    {
        public void Main()
        {
            RaycastHit hit;
            if (Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition),
out hit, maxDistance: 100.0f))
            {

            }
        }
    }
}
```

Теперь переменные out можно объявлять в списке аргументов в вызове метода, не записывая отдельный оператор объявления.

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleRaycast
    {
        public void Main()
        {
            if (Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition),
out var hit, maxDistance: 100.0f))
            {

            }
        }
    }
}
```

Плюсы:

- Код проще читать.
- Переменная out объявляется при использовании, а не в другой, вышестоящей строке.
- Назначать начальное значение не нужно.
- Объявляя переменную out, когда она используется при вызове метода, ее нельзя случайно использовать прежде, чем она будет назначена.

При деконструкции кортежа или вызове метода с параметрами out часто требуется определить переменную, которую вы не планируете использовать и значение которой не важно. Для работы в таких сценариях в C# реализована поддержка пустых переменных. Пустая переменная представляет собой доступную только для записи переменную с именем _ (знак подчеркивания). Вы можете назначить одной переменной все значения, которые не потребуются в дальнейшем. Пустая

переменная является аналогом неприсвоенной переменной и не может использоваться в коде где-либо, за исключением оператора присваивания.

- Пустые переменные поддерживается в следующих случаях.
- При деконструкции кортежей или пользовательских типов.
- При вызове методов с параметрами out.
- В операции сопоставления шаблонов с выражениями is и switch.
- В качестве автономного идентификатора в тех случаях, когда требуется явно идентифицировать значение присваивания как пустую переменную.

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleRaycast
    {
        public void Main()
        {
            if (Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition),
out _, maxDistance: 100.0f))
            {

            }
        }
    }
}
```

Еще пример

```
namespace Geekbrains
{
    public sealed class ExampleRaycast
    {
        public void Main()
        {
            Example(out var x, out var y);
            Example(out _, out var y2);
            Example(out var x2, out _);
            Example(out _, out _);
        }

        private void Example(out int x, out int y)
        {
            x = 10;
            y = 10;
        }
    }
}
```

Локальные переменные и возвращаемые значения Ref

Добавление локальных переменных и возвращаемых значений ref дает возможность использовать более эффективные алгоритмы, поскольку избавляет от необходимости многократно копировать значения или выполнять операции разыменования.

```
using UnityEngine;

namespace Geekbrains
{
```

```
    public sealed class ExampleRefLocal
    {
        public void Main()
        {
            int temp = 42;
            ref int tempReference = ref temp;
            Debug.Log(temp); // 42
            tempReference = 123;
            Debug.Log(temp); // 123
            temp = 321;
            Debug.Log(tempReference); // 321
            int tempSecond = 24;
            tempReference = ref tempSecond;
            Debug.Log(tempReference); // 24
        }
    }
}
```

Язык C# включает правила, которые защищают вас от неправильного использования локальных переменных и возвращаемых значений ref:

- Необходимо добавить ключевое слово ref в сигнатуру метода и все инструкции return в методе.
- Это позволяет уточнить, что метод возвращает значение по ссылке, во всех местах.
- Объект ref return может быть назначен переменной-значению или переменной ref.
- Вызывающий объект определяет, копируется ли возвращаемое значение. Пропуск модификатора ref при присваивании возвращаемого значения указывает, что вызывающий объект хочет получить копию значения, а не ссылку на хранилище.
- Присвоить локальной переменной ref стандартное возвращаемое значение метода невозможно.
- Это запрещает использовать операторы вида ref int i = sequence.Count();
- Переменную ref невозможно возвращать переменной, которая продолжает существовать даже после того, как метод будет выполнен.
- Это означает невозможность возвращения ссылки на локальную переменную или переменную с аналогичной областью.
- Возвращаемые значения и локальные переменные ref не могут использоваться с асинхронными методами.
- На момент, когда асинхронный метод возвращает значение, компилятору неизвестно, присвоено ли переменной, на которую указывает ссылка, окончательное значение.

```
using System;
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleRefLocal
    {
        public void Main()
        {
            int[] numbers = { 1, 2, 3, 4, 5, 6, 7 };
            ref int numberRef = ref Find(4, numbers); // ищем число 4 в массиве
            numberRef = 9; // заменяем 4 на 9
            Debug.Log(numbers[3]); // 9
        }
        private ref int Find(int number, int[] numbers)
        {
            for (var i = 0; i < numbers.Length; i++)
            {
                if (numbers[i] == number)
```

```
                {
                    return ref numbers[i]; // возвращаем ссылку на адрес, а не само
значение
                }
            }
            throw new IndexOutOfRangeException("число не найдено");
        }
    }
}
```

Ключевое in инициирует передачу аргументов по ссылке. В результате этот формальный параметр становится псевдонимом для аргумента, который должен представлять собой переменную. Другими словами, любая операция в параметре осуществляется с аргументом. Оно аналогично ключевым словам ref и out, за исключением того, что аргументы in не могут быть изменены вызываемым методом. В то время как аргументы ref могут быть изменены, аргументы out должны быть изменены вызывающим объектом, и эти изменения отслеживаются в вызывающем контексте.

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleRefLocal
    {
        public void Main()
        {
            int readonlyArgument = 44;
            InArgExample(readonlyArgument);
            Debug.Log(readonlyArgument); // value is still 44

            void InArgExample(in int number)
            {
                // Uncomment the following line to see error CS8331
                //number = 19;
            }
        }
    }
}
```

Локальные функции

Начиная с версии 7.0 в языке C# поддерживаются локальные функции. Локальные функции представляют собой частные методы типа, вложенные в другой элемент. Они могут вызываться только из того элемента, в который вложены.

```
using System;
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleLocalFunctions
    {
        private void GetAge()
        {
            var ageInput = "17";

            if (!IsAdult(ageInput))
            {
                Debug.Log("Регистрация разрешена только совершеннолетним");
            }
```

```
            bool IsAdult(string value) // Локальная функция
            {
                if (!Int32.TryParse(value, out var age)) throw new
Exception(@"Возраст введен некорректно");
                if (age > 18)
                {
                    return true;
                }
                return false;
            }
        }
    }
}
```

Ниже приведен список наиболее важных аспектов локальных функций в C#:

- Локальные функции могут определять блоки итераторов.
- Локальные функции полезны для неотложенной (eager) проверки предусловий в асинхронных методах и блоках итераторов.
- Локальные функции могут быть рекурсивными.
- Локальные функции не аллоцируют в куче, если не происходит преобразование их в делегаты.
- Локальные функции немного более эффективны, чем анонимные функции из-за отсутствия накладных расходов вызовов делегата.
- Локальные функции могут быть объявлены после оператора return, что позволяет отделить основную логику метода от вспомогательной.
- Локальные функции могут «скрыть» функцию с тем же именем, объявленным во внешней области видимости.
- Локальные функции могут быть асинхронными и/или небезопасными (unsafe); другие модификаторы не допускаются.
- Локальные функции не могут иметь атрибуты.

Начиная с версии C# 8.0 можно определять статические локальные функции. Их особенностью является то, что они не могут обращаться к переменным окружения, то есть метода, в котором статическая функция определена.

Замыкание

Язык C# даёт нам возможность пользоваться замыканиями — мощным механизмом, который позволяет анонимным методам и лямбдам захватывать свободные переменные в своём лексическом контексте.

```
using System;
using System.Collections.Generic;
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleClosure
    {
        public void Main()
        {
            var actions = new List<Action>();
            for (var i = 0; i < 100; i++)
            {
                actions.Add(() => Debug.Log(i));
            }
```

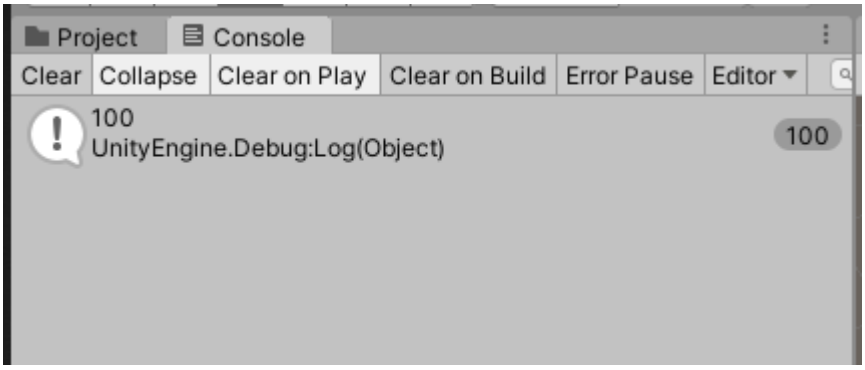
```
            foreach (var action in actions)
            {
                action();
            }
        }
    }
}
```

Вызовем метод Main().

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class Test : MonoBehaviour
    {
        private void Start()
        {
            new ExampleClosure().Main();
        }
    }
}
```

Результат получился неожиданным



Немного перепишем метод Main().

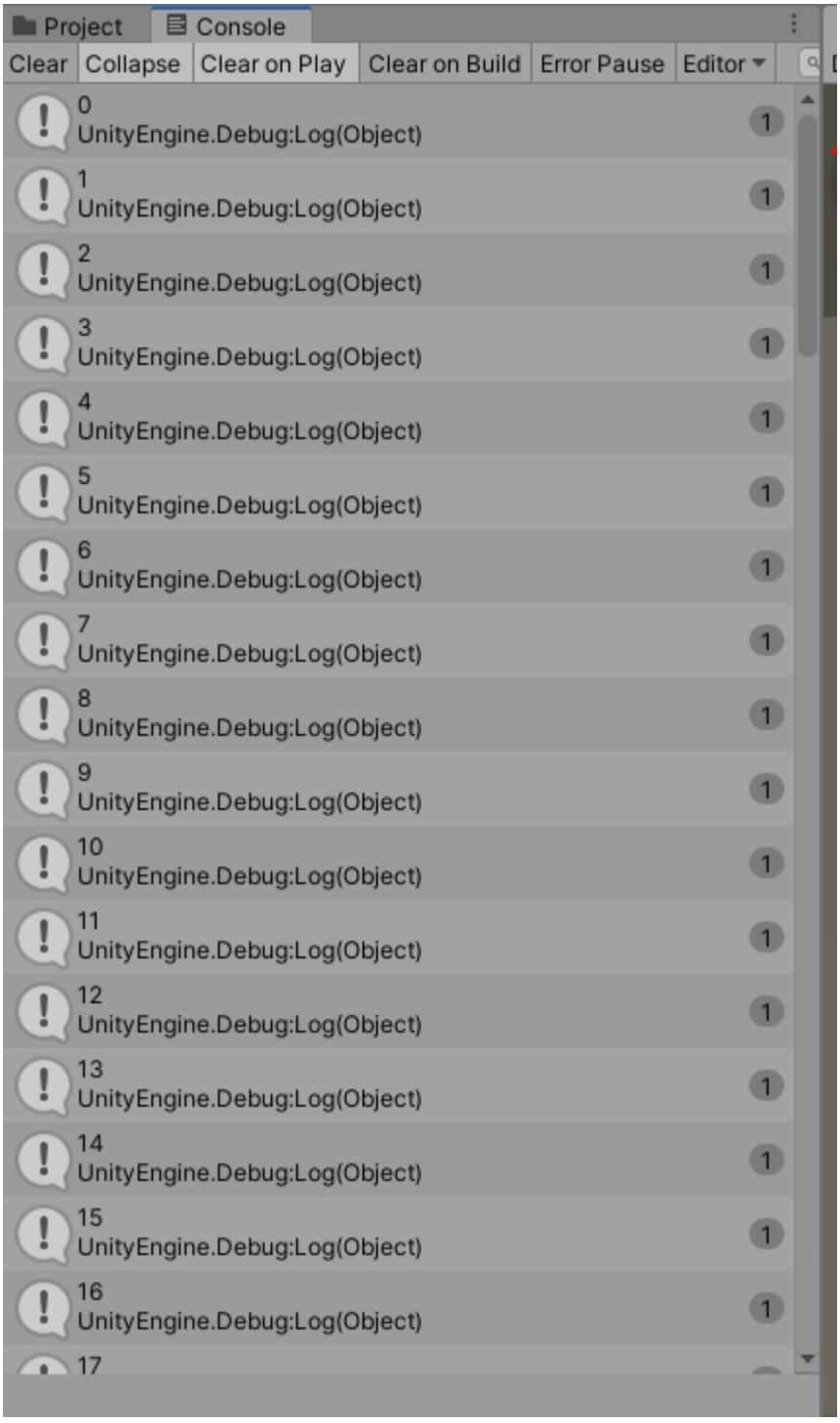
```
using System;
using System.Collections.Generic;
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleClosure
    {
        public void Main()
        {
            var actions = new List<Action>();
            for (var i = 0; i < 100; i++)
            {
                var tempI = i;
                actions.Add(() => Debug.Log(tempI));
            }

            foreach (var action in actions)
            {
                action();
```

```
            }
        }
    }
}
```

Теперь результат будет другим



Посмотрим, во что компилятор развернёт такую конструкцию:

```
public void Run()
{
    var list = new List<Action>();
    DisplayClass c = new DisplayClass();
```

```
    for (c.i = 0; c.i < 3; c.i++)
    {
        list.Add(c.Action);
    }

    foreach (Action action in list)
    {
        action();
    }
}

private sealed class DisplayClass
{
    public int i;

    public void Action()
    {
        Debug.Log(i);
    }
}
```

Следующие правила применимы к области действия переменной в лямбда-выражениях.

- Захваченная переменная не будет уничтожена сборщиком мусора до тех пор, пока делегат, который на нее ссылается, не перейдет в статус подлежащего уничтожению при сборке мусора.
- Переменные, представленные в лямбда-выражении, невидимы в заключающем методе.
- Лямбда-выражение не может непосредственно захватывать параметры in, ref или out из заключающего метода.
- Оператор return в лямбда-выражении не вызывает возврат значения заключающим методом.
- Лямбда-выражение не может содержать операторы goto, break или continue, если целевой объект этого оператора перехода находится за пределами блока лямбда-выражения. Если целевой объект находится внутри блока, использование оператора перехода за пределами лямбда-выражения также будет ошибкой.

Паттерны switch

Выражения switch

Часто инструкция switch возвращает значение в каждом из блоков case. Выражения switch позволяет использовать более краткий синтаксис выражения. В нем меньше повторяющихся ключевых слов case и break, а также меньше фигурных скобок

```
using System;
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExamplePatternSwitch
    {
        private enum Rainbow
        {
            None   = 0,
            Red    = 1,
            Orange = 2,
            Yellow = 3,
            Green  = 4,
            Blue   = 5,
```

```
            Indigo = 6,
            Violet = 7
        }

        private void Main()
        {
            Debug.Log(Select(Rainbow.Blue));
        }

        private string Select(Rainbow rainbow)
        {
            switch (rainbow)
            {
                case Rainbow.Red: return "Red";
                case Rainbow.Orange: return "Orange";
                case Rainbow.Yellow: return "Yellow";
                case Rainbow.Green: return "Green";
                case Rainbow.Blue: return "Blue";
                case Rainbow.Indigo: return "Indigo";
                case Rainbow.Violet: return "Violet";
                default: throw new ArgumentException("Нет такого цвета");
            }
        }
    }
}
```

Начиная с версии C# 8.0

```
private string Select(Rainbow rainbow)
{
    return rainbow switch
    {
        Rainbow.Red => "Red",
        Rainbow.Orange => "Orange",
        Rainbow.Yellow => "Yellow",
        Rainbow.Green => "Green",
        Rainbow.Blue => "Blue",
        Rainbow.Indigo => "Indigo",
        Rainbow.Violet => "Violet",
        _ => throw new ArgumentException("Нет такого цвета")
    };
}
```

Здесь представлено несколько улучшений синтаксиса:

- Переменная расположена перед ключевым словом switch. Другой порядок позволяет визуально легко отличить выражение switch от инструкции switch.
- Элементы case и : заменяются на =>. Это более лаконично и интуитивно понятно.
- Случай default заменяется пустой переменной _.
- Тексты являются выражениями, а не инструкциями.

Шаблоны свойств

Шаблон свойств позволяет сопоставлять свойства исследуемого объекта.

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExamplePatternSwitch
```

```
    {
        private sealed class Student
        {
            public string Name { get; set; }
            public string EducationalInstitution { get; set; }
            public string AcademicPerformance { get; set; }
        }

        private void Main()
        {
            Debug.Log(Select(new Student { EducationalInstitution = "Geekbrains"}));
        }

        private string Select(Student student)
        {
            return student switch
            {
                { EducationalInstitution: "Institute", AcademicPerformance:
"Excellent" } => "Здравствуйте",
                { EducationalInstitution: "College" } => "Привет!",
                { EducationalInstitution: "School" } => "Здоров",
                { } => "Hello world!",
                _ => "Not Found"
            };
        }
    }
}
```

Шаблоны кортежей

Некоторые алгоритмы зависят от нескольких наборов входных данных. Шаблоны кортежей позволяют переключаться между несколькими значениями, выраженными как кортежи.

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExamplePatternSwitch
    {
        private enum Gender
        {
            None      = 0,
            Male      = 1,
            Female    = 2,
            Indefined = 3
        }

        private void Main()
        {
            Debug.Log(Select(Gender.Female, 12));
        }

        private string Select(Gender gender, int age)
        {
            return (gender, age) switch
            {
                (Gender.Female, 14) => "Девочка",
                (Gender.Female, 18) => "Девушка",
                (Gender.Female, 30) => "Женщина",
                (Gender.Female, 70) => "Бабушка",
```

```
                _ => "Не понятно"
            };
        }
    }
}
```

Позиционный паттерн

Позиционный паттерн применяется к типу, у которого определен метод деконструктора.

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExamplePatternSwitch
    {
        private enum Gender
        {
            None      = 0,
            Male      = 1,
            Female    = 2,
            Indefined = 3
        }
        private sealed class People
        {
            public string Name { get; set; }
            public Gender Gender { get; set; }
            public int Age { get; set; }

            public void Deconstruct(out string name, out Gender gender, out int age)
            {
                name = Name;
                gender = Gender;
                age = Age;
            }
        }
        private void Main()
        {
            Debug.Log(Select(new People()));
        }

        private string Select(People people)
        {
            return people switch
            {
                ("Lera", Gender.Female, 18) => "Female",
                ("Roman", Gender.Male, _) => "Male",
                ("Ilya", Gender.Indefined, _) => "Dean",
                (_, _, 17) => "Minor",
                (_, Gender.Indefined, _) => "Indefined",
                _ => "Not recognized"
            };
        }
    }
}
```

Объявления using

Объявление using — это объявление переменной, которому предшествует ключевое слово using. Оно сообщает компилятору, что объявляемая переменная должна быть удалена в конце области видимости. Для примера рассмотрим следующий код, в котором записывается текстовый файл:

```
using System.Collections.Generic;

namespace Geekbrains
{
    public sealed class ExampleUsing
    {
        private int WriteLinesToFile(IEnumerable<string> lines)
        {
            using var file = new System.IO.StreamWriter("WriteLines2.txt");
            // Notice how we declare skippedLines after the using statement.
            int skippedLines = 0;
            foreach (string line in lines)
            {
                if (!line.Contains("Second"))
                {
                    file.WriteLine(line);
                }
                else
                {
                    skippedLines++;
                }
            }
            // Notice how skippedLines is in scope here.
            return skippedLines;
            // file is disposed here
        }
    }
}
```

Данный функционал появился в C# 8.0

Span

Span<T> — это структура ссылки, которая выделяется в стеке, а не в управляемой куче. Типы структур ref имеют ряд ограничений, чтобы гарантировать, что их нельзя повысить до управляемой кучи, в том числе что они не могут быть упакованы, они не могут быть назначены переменным типа Object, dynamic или типу интерфейса, они не могут быть полями в ссылочном типе и не могут использоваться в границах await и yield. Кроме того, вызовы двух методов, Equals(Object) и GetHashCode, вызывают NotSupportedException.

Lazy

Отложенная инициализация объекта означает, что его создание откладывается до первого использования. Отложенная инициализация в основном используется для повышения производительности, предотвращения непроизводительна вычислений и сокращения требований к памяти программы. Ниже приведены наиболее распространенные сценарии.

При наличии объекта, создание которого требует много ресурсов и который, возможно, не будет использоваться программой. Например, предположим, что в памяти находится объект Customer, у которого есть свойство Orders, содержащее большой массив объектов Order, инициализация которых

требует подключения к базе данных. Если пользователь никогда не отображает массив Orders и не использует его данные в расчетах, то нет смысла использовать системную память или такты процессора для создания этого массива. Используя Lazy<Orders>, чтобы объявить отложенную инициализацию объекта Orders, можно избежать расхода системных ресурсов на неиспользуемый объект.

При наличии объекта, требующее много ресурсов создание которого желательно отложить до завершения других ресурсоемких операций. Например, пусть программа во время запуска загружает несколько экземпляров объекта, но только часть из них требуется сразу. Можно повысить быстродействие программы при запуске, отложив инициализацию временно ненужных объектов до того, как они понадобятся.

Хотя вы можете написать свой код для выполнения отложенной инициализации, вместо этого рекомендуется использовать тип Lazy<T>. Тип Lazy<T> и связанные с ним типы также поддерживают безопасность потоков и обеспечивают согласованную политику распространения исключений.

Пример использования отложенной инициализации на примере создания паттерна одиночка

```
using System;
using UnityEngine;

namespace Geekbrains
{
    public sealed class ExampleLazy
    {
        public sealed class Singleton
        {
            private Singleton() { }

            private static readonly Lazy<Singleton> _instance = new
Lazy<Singleton>(() => new Singleton());

            public static Singleton Instance => _instance.Value;

            public void Test()
            {
                Debug.Log("Hello World");
            }
        }

        public void Main()
        {
            Singleton.Instance.Test();
        }
    }
}
```

Индексы и диапазоны

В C# 8.0 была добавлена новая функциональность - индексы и диапазоны, которые упрощают получение из массивов подмассивов.

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class Example
    {
```

```
        public void Main()
        {
            Index indexFirst = 2;     // третий элемент
            Index indexSecond = ^2;    // предпоследний элемент

            int[] number = { 4, 8, 15, 16, 23, 42 };
            int selected1 = number[indexFirst];    // 15
            int selected2 = number[indexSecond];    // 23
            Debug.Log(selected1);
            Debug.Log(selected2);
        }
    }
}
```

С помощью специального оператора ^ можно задать индекс относительно конца последовательности.

Диапазон

Диапазон представляет часть последовательности, которая ограничена двумя индексами. Начальный индекс включается в диапазон, а конечный индекс НЕ входит в диапазон. Для определения диапазона применяется оператор ..:

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class Example
    {
        public void Main()
        {
            int[] numbers = { 4, 8, 15, 16, 23, 42 };
            string[] numbersRange = numbers[1..4]; // получаем 2, 3 и 4-й элементы из
массива
            foreach(var number in numbersRange)
            {
                Debug.Log(number);
            }
        }
    }
}
```

Еще пример

```
namespace Geekbrains
{
    public sealed class Example
    {
        public void Main()
        {
            int[] numbers = { 4, 8, 15, 16, 23, 42 };
            string[] numberRange1 = numbers[^2..];         // два последних
            string[] numberRange2 = numbers[..^1];     // начиная с предпоследнего
            string[] numberRange3 = numbers[^3..^1];       // два начиная с
предпоследнего
        }
    }
}
```

Ковариантность и контравариантность

Термины "ковариантность" и "контрвариантность" относятся к возможности использовать более производный (более конкретный) или менее производный (менее конкретный) тип, чем задано изначально. Параметры универсальных типов поддерживают ковариантность и контрвариантность и обеспечивают большую гибкость в назначении и использовании универсальных типов. Ниже приведены определения терминов "ковариантность", "контрвариантность" и "инвариантность" в контексте системы типов. В этих примерах предполагается наличие базового класса с именем Base и производного класса с именем Specific.

Covariance

Позволяет использовать тип с большей глубиной наследования, чем задано изначально.

Экземпляр IEnumerable<Specific> можно присвоить переменной типа IEnumerable<Base>.

Contravariance

Позволяет использовать более универсальный тип (с меньшей глубиной наследования), чем заданный изначально.

Экземпляр Action<Base> можно присвоить переменной типа Action<Specific>.

Invariance

Означает, что можно использовать только изначально заданный тип. Таким образом, параметр инвариантного универсального типа не является ни ковариантным, ни контравариантным.

Экземпляр List<Base> нельзя присвоить переменной типа List<Specific>, и наоборот.

```
namespace Geekbrains
{
    public sealed class ExampleCovariance
    {
        public void Main()
        {
            Animals animals = new Cat();

            ITestCovariance<Animals> testInvariance = new TestCovariance<Animals>();
// Инвариантность: позволяет использовать только заданный тип

            ITestCovariance<Animals> testCovariance = new TestCovariance<Cat>(); //
Ковариантность: позволяет использовать более конкретный тип, чем заданный изначально
<out T>

            ITestContravariance<Cat> testContravariance = new
TestContravariance<Animals>(); // Контравариантность: позволяет использовать более
универсальный тип, чем заданный изначально <in T>
        }
    }

    public class Animals
    {

    }

    public class Cat : Animals
```

```
    {

    }

    public interface ITestCovariance<out T>
    {
        T Test(int t);
    }

    public class TestCovariance<T> : ITestCovariance<T>
    {
        public T Test(int t)
        {
            return default;
        }
    }

    public interface ITestContravariance<in T>
    {
        void Test(T t);
    }

    public class TestContravariance<T> : ITestContravariance<T>
    {
        public void Test(T t)
        {

        }
    }
}
```

Ковариантными и контрвариантними могут быть только обобщенные интерфейсы и делегаты

Практическое задание

1. Доделать проект под архитектуру MV
2. *Разработать ковариантный и контрвариантный делегат
3. *Передать данные об полученном бонусе используя кортежи
4. *Добавить в игру регистрацию игрока и сохранение его прогресса

Дополнительные материалы

1. Ковариантность и контравариантность (Википедия)
2. https://habr.com/ru/post/218753/
3. https://andrewlock.net/deconstructors-for-non-tuple-types-in-c-7-0/
4. https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/nameof
5. https://docs.microsoft.com/ru-ru/dotnet/csharp/tuples
6. https://aakinshin.net/ru/posts/closures/

7. https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/ref?view=netcore-3.1#ref-struct-types
8. https://docs.microsoft.com/ru-ru/dotnet/framework/performance/lazy-initialization
9. https://docs.microsoft.com/ru-ru/dotnet/csharp/whats-new/csharp-8

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Язык программирования C# 7 и платформы .NET и .NET Core | Троелсен Эндрю, Джепикс Филипп – 2018 г.
2. Unity в действии. Мультиплатформенная разработка на C# | Джозеф Хокинг - 2016 г.
3. https://docs.unity3d.com/Manual/index.html
4. MSDN.