GeekBrains

Основы C# в Unity

Основы ООП в Unity

На этом уроке:

1. Вы узнаете, что такое класс и структура и чем они отличаются от объекта.
2. Разберёте, из чего состоит класс и структура.
3. Узнаете, на чём базируется ООП.
4. Познакомитесь с фишками языка C#.

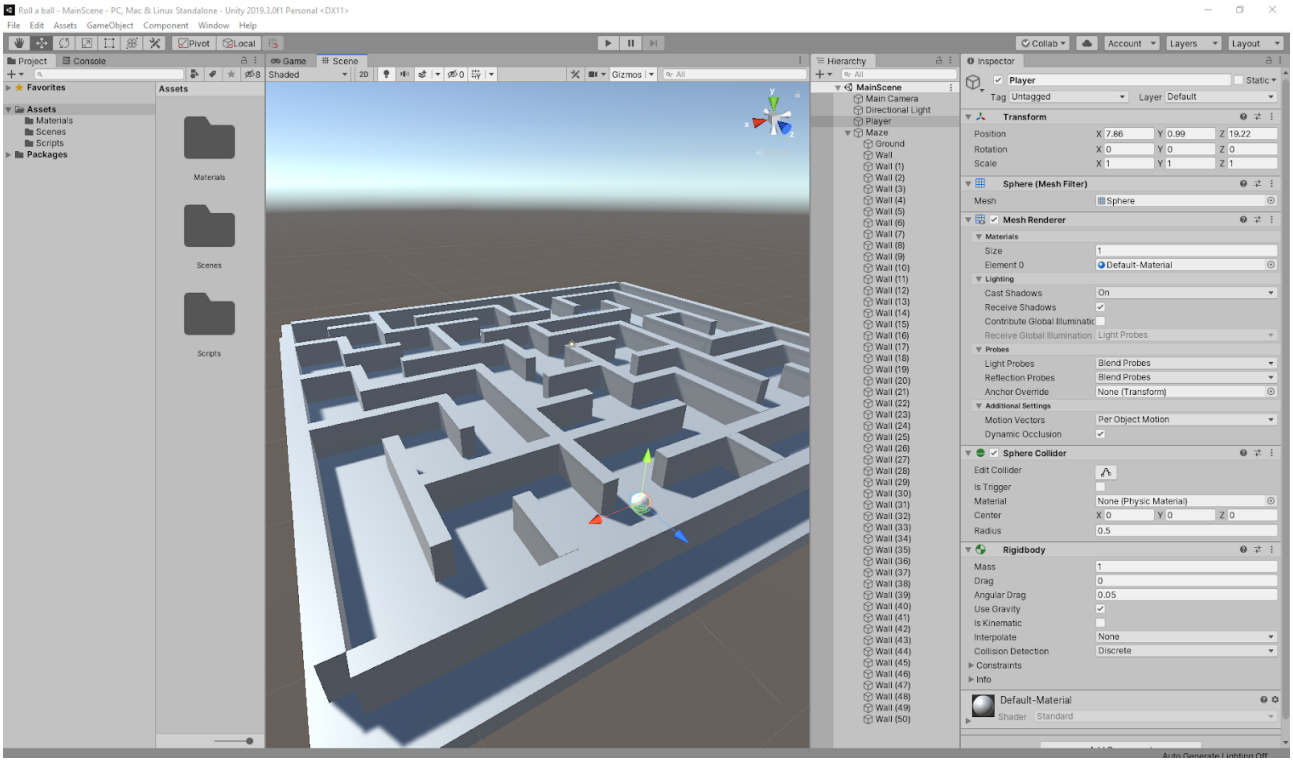
Краткое описание урока

Структуры и их особенности. Отличие класса и структуры, класса и объекта. Конструкторы, поля, свойства и методы. Наследование, инкапсуляция и полиморфизм.

Оглавление

Подготовка к игре

Изучение данного курса мы начнем с игры Roll a Ball. Для начала нам нужно подготовить сцену. Создадим небольшой лабиринт, камеру, источник освещения и главного персонажа.



В игре будет уровень в виде лабиринта. По уровню будут расставлены бонусы, которые необходимо собрать. А усложнят задачу находящиеся в лабиринте ловушки.

Типы данных

В языке C# существуют 2 типа данных. Ссылочный и значимый. Вот несколько примеров:

Типы значений:

1. перечисления enum, например, LightType и т. д.
2. структуры (struct), например, byte, int, float, bool, Vector3, Quaternion и т. д.

Ссылочные типы:

1. Тип object
2. Тип string
3. Классы (class)
4. Интерфейсы (interface)
5. Делегаты (delegate)

Важно понимать между ними различие! В C# устройство памяти следующие, есть стек и управляемая куча. Значимые типы данных создаются в стеке, а ссылочные в куче.

Объект

Объектом называется экземпляр класса или структуры.

Структуры

Структуры относятся к типам значений. Структуры необходимы, когда нужно создавать часто экземпляры объектов

```
namespace Geekbrains
{
    public struct Vector
    {
        public float x;
        public float y;
    }
}
```

Классы

Класс в отличие от структуры является ссылочным типом данных. Класс также, как и структура описывает свойства и поведение объекта. Для создания экземпляра класса нам необходимо использовать ключевое слово **new**. Например, есть класс Player

```
namespace Geekbrains
{
    public class Player
    {

    }
}
```

Пример создания экземпляра класса Player

```
Player player = new Player();
```

Говоря про класс **Player**, мы подразумеваем все объекты, которые могут быть созданы. Говоря про объект **player**, мы имеем в виду конкретный экземпляр объекта, который хранится в памяти.

Основные конструкции

Любой класс либо структура могут содержать в себе следующий набор элементов

1. Поля
2. Свойства
3. Конструктор
4. Метод
5. Перегруженный оператор

Данный список мы будем дополнять по мере прохождения курса.

Поле

Поля объектов необходимы для хранения свойств объекта. Сигнатура у поля следующая

[модификатор доступа] [тип данных] [название поля]

```
namespace Geekbrains
{
    public class Player
    {
        public float speed;
    }
}
```

В примере выше представлено поле speed класса Player, с модификатором доступа public и тип данных float. Ниже представлены все модификаторы доступа версии языка C# 7.

Модификаторы доступа

В C# .Net существует 6 модификаторов доступа:

- **public** – публичный, общедоступный класс или член класса. Он доступен из любого места в коде, а также из других программ и сборок;
- **private** – закрытый класс или член класса, полная противоположность модификатору **public**. Доступен только из кода в том же классе или контексте;
- **protected** – доступен из любого места в текущем классе или в производных классах;
- **internal** – класс и члены класса с подобным модификатором доступны из любого места кода в той же сборке. Но он недоступен для других программ и сборок (как модификатор **public**);
- **protected internal** – совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.
- **private protected** – комбинация ключевых слов private protected является модификатором доступа к члену. К члену private protected имеют доступ типы, производные от содержащего класса, но только в пределах содержащей сборки. Сравнение модификатора private protected с другими модификаторами доступа (https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/private-protected).

Если мы явно не указываем модификатор доступа, он выставляется автоматически. Для класса и структуры это **internal**, поля класса и функции – **private**.

Конструктор

Конструктор – это специальный метод, который вызывается при создании экземпляра объекта. В **.Net Framework**, если в созданном вами классе нет описанного вами конструктора, создается конструктор без параметров, который заполняет поля объекта данными по умолчанию (**0**, **false**, **null**). Чтобы при создании объект был заполнен данными, добавляем конструктор с параметрами.

```
namespace Geekbrains
{
    public class Player
    {
        public float speed;

        public Player()
        {
            speed = 3.0f;
        }
```

```
        public Player(float speed) : this()
        {
            this.speed = speed;
        }
    }
}
```

В примере выше перегружен конструктор по умолчанию и определен конструктор с входящим параметром float.

Стоит заметить, что для структур нельзя перегружать конструктор по умолчанию и нужно в конструкторе проинициализировать все его поля.

```
namespace Geekbrains
{
    public struct Vector
    {
        public float x;
        public float y;

        public Vector(float x, float y)
        {
            this.x = x;
            this.y = y;
        }
    }
}
```

Статический конструктор

Кроме обычных конструкторов у класса также могут быть статические конструкторы. Они выполняются при самом первом создании объекта данного класса или первом обращении к его статическим членам (если таковые имеются). И у статического конструктора не указывается модификатор доступа. Не явно будет выставлен модификатор public

```
namespace Geekbrains
{
    public class Player
    {
        public static float speed;

        static Player()
        {
            speed = 3.0f;
        }
    }
}
```

Ключевое свойство readonly

Ключевое readonly слово означает, что присвоить значение данному полю можно будет только при инициализации или в конструкторе

```
public readonly float speed = 3.0f;
```

readonly так же могут быть и структуры, при условии что их поля тоже будут readonly.

```
namespace Geekbrains
{
    public readonly struct Vector
    {
        public readonly float x;
        public readonly float y;

        public Vector(float x, float y)
        {
            this.x = x;
            this.y = y;
        }
    }
}
```

Константы

Константы предназначены для описания таких значений, которые не должны изменяться в программе. Для определения констант используется ключевое слово const

```
namespace Geekbrains
{
    public class Player
    {
        public const float Speed = 3.0f;

        private void Start()
        {
            const string name = "Roman";
        }
    }
}
```

Константы должны быть проинициализированы при определении. Так же можно определять локальные константы

Методы

Для доступа к закрытым данным можно использовать открытые методы:

```
using UnityEngine;


namespace Geekbrains
{
    public class Player
```

```
    {
        private float _speed;

        public float GetSpeed()
        {
            return _speed;
        }

        public void SetSpeed(float speed)
        {
            if (speed > 0)
            {
                _speed = speed;
            }
        }
    }
}
```

Expression-Bodied Methods

С версии C# 6.0 появилась новая функциональность - встроенные методы, или Expression-Bodied Methods. Они позволяют применять лямбда-выражения для сокращенного написания методов в одну строку. Например вот так можно сократить метод GetSpeed из класса Player.

```
public float GetSpeed() => _speed;
```

Свойства

С помощью свойств можно задавать значения закрытым полям объекта. Технология свойств объединяет в себе поле и метод. Вместо того, чтобы создавать несколько методов для записи или чтения данных из полей объекта, можно объединить эти действия в одном свойстве:

```
namespace Geekbrains
{
    public class Player
    {
        private float _speed;
        public float Speed
        {
            get => _speed;
            set
            {
                if (value > 0)
                {
                    _speed = value;
                }
            }
        }
    }
}
```

При описании свойств используются аксессоры доступа **get** и **set**. С их помощью программист управляет возможностями читать или записывать данные. Можно применять модификаторы доступа не только ко всему свойству, но и к отдельным блокам – **get** или **set**. Модификатор можно применить только к одному из блоков:

```
namespace Geekbrains
{
    public class Player
    {
        private float _speed;

        public float Speed
        {
            private get => _speed;
            set
            {
                if (value > 0)
                {
                    _speed = value;
                }
            }
        }
    }
}
```

Автоматические свойства

С версии C# 3.0 появилась возможность реализовать очень простые свойства, не прибегая к явному определению переменной, которой они управляют. Вместо этого базовую переменную для свойства автоматически предоставляет компилятор. Такое свойство называется автоматически реализуемым и принимает следующую общую форму:

```
тип имя { get; set; }
```

Здесь тип обозначает конкретный тип свойства, а имя – присваиваемое свойству имя. Обратите внимание на то, что после обозначений аксессоров **get** и **set** сразу же следует точка с запятой, а тело у них отсутствует. Такой синтаксис предписывает компилятору автоматически создать для хранения значения переменную, иногда еще называемую поддерживающим полем. Такая переменная недоступна непосредственно и не имеет имени, но может быть доступна через свойство.

Пример описания автоматического свойства:

```
public float Speed { get; set; }
```

С версии C# 6.0 можно проинициализировать автосвойство:

```
public float Speed { get; set; } = 3.0f;
```

В C# 5.0, чтобы автосвойство было доступным для установки только из класса, необходимо указать **private set**. С версии C# 6.0 не обязательно писать **private set** – можно оставить только выражение **get**.

Для хранения значения этого свойства для него неявно будет создаваться поле с модификатором **readonly**. Следует учитывать, что подобные get-свойства можно установить либо из конструктора класса, либо при инициализации свойства.

Перегрузка операторов

Покажем перегрузку операторов на примере класса **Vector**. Перегрузка операторов позволяет использовать пользовательские классы с привычными операторами **+, -, *, =** и другими.

Ключевое слово **implicit** служит для объявления неявного оператора преобразования пользовательского типа. Этот оператор обеспечивает неявное преобразование между пользовательским типом и другим типом, если при преобразовании исключается утрата данных
explicit - Ключевое слово для объявления явного оператора преобразования. Значит, мы указываем на потерю данных.

```
namespace Geekbrains
{
    public struct Vector
    {
        public double X { get; private set; }

        public double Y { get; private set; }

        public Vector(double x, double y)
        {
            X = x;
            Y = y;
        }

        public static explicit operator Vector(double x) => new Vector(x, x);

        public static implicit operator double(Vector x) => x.X;

        public override string ToString() => $"X= {X} Y= {Y}";

        public static Vector operator +(Vector v1, Vector v2)
        {
            Vector res = new Vector
            {
                X = v1.X + v2.X,
                Y = v1.Y + v2.Y
            };
            return res;
        }

        public static Vector operator -(Vector v1, Vector v2)
        {
            Vector res = new Vector
            {
                X = v1.X - v2.X,
                Y = v1.Y - v2.Y
            };
            return res;
        }

        public static Vector operator -(Vector v1)
        {
            Vector res = new Vector
            {
                X = -1 * v1.X,
                Y = -1 * v1.Y
            };
            return res;
        }
    }
}
```

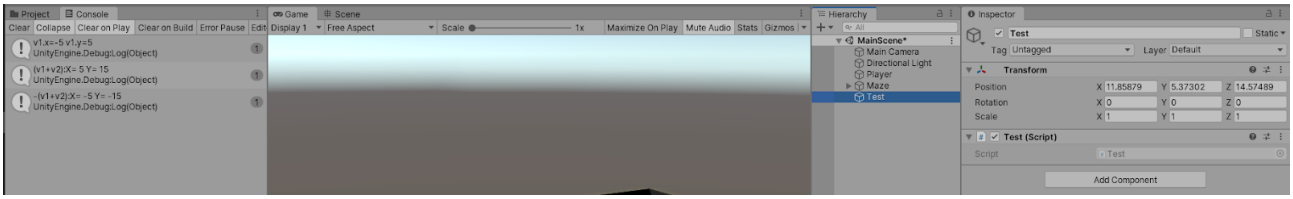
В примере продемонстрировано, как можно перегрузить операции для новых типов данных. Эта возможность позволяет упростить дальнейшее программирование, так как теперь операции сложения, вычитания и присваивания происходят как над единым объектом.

Для демонстрации примере создадим класс Test

```
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            Vector v1 = new Vector(-5, 5);

            Vector v2 = (Vector) 10;
            Vector v3 = v1 + v2;

            Debug.Log($"v1.x={v1.X} v1.y={v1.Y}");
            Debug.Log($"(v1+v2):{v3}");
            Debug.Log($"-(v1+v2):{-v3}");
        }
    }
}
```

На сцене создадим пустой GameObject назовем Test его и добавим на него класс Test

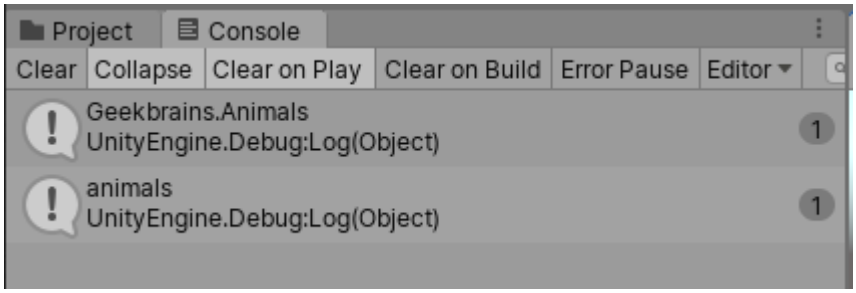
Результат примера:



nameof и typeof

Оператор nameof позволяет название объекта, метода, свойства и т.д получит в виде строки. Оператор typeof позволяет получить тип данных.

```
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            Animals animals = new Animals();

            Debug.Log(typeof(Animals));
            Debug.Log(nameof(animals));
        }
    }
}
```



Объектно-ориентированное программирование

C# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.

Описанием объекта является класс, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек (фактически экземпляр данного класса) будет представлять объект этого класса.

Инкапсуляция

Инкапсуляция — это механизм программирования, объединяющий вместе код и данные, которыми он манипулирует, исключая как вмешательство извне, так и неправильное использование данных. В объектно-ориентированном языке данные и код могут быть объединены в совершенно автономный черный ящик. Внутри такого ящика находятся все необходимые данные и код. Когда код и данные связываются вместе подобным образом, создается объект. Иными словами, объект — это элемент, поддерживающий инкапсуляцию. Т.е. инкапсуляция представляет собой способности языка скрывать излишние детали реализации от пользователя объекта.

Наследование

Наследование представляет собой процесс, в ходе которого один объект приобретает свойства другого.

В языке C# наследуемый класс называется базовым, а наследующий – производным. Следовательно, производный класс представляет собой специализированный вариант базового класса. Он наследует все переменные, методы, свойства и индексаторы, определяемые в базовом классе и может добавить к ним собственные элементы.

Наследование является одним из трех основополагающих принципов объектно-ориентированного программирования, поскольку оно допускает создание иерархических классификаций. Благодаря наследованию можно создать общий класс, в котором определяются характерные особенности, присущие множеству связанных элементов. От этого класса могут затем наследовать другие, более конкретные классы, добавляя в него свои индивидуальные особенности.

В C# (точнее, в .Net Framework) все объекты наследуются от базового класса **object**. В практическом смысле мы можем присваивать переменной класса **object** любое значение.

```
Object a=new Object();
a=10;
a=3.14;
a="Строка";
a=new Random();
```

Хотя такой способ существует и может быть использован при программировании, разработчики стараются избегать его, так как приходится заботиться о том, какие данные хранятся в переменных типа **object**. При данном подходе вы столкнетесь с такими явлениями, как упаковка (boxing) и распаковка (unboxing). Первая предполагает преобразование объекта значимого типа (например, типа **int**) к типу **object**. При упаковке общеязыковая среда **CLR** обертывает значение в объект типа **System.Object** и сохраняет его в управляемой куче (хипе). Распаковка (unboxing), наоборот, предполагает преобразование объекта типа **object** к значимому типу. Упаковка и распаковка ведут к снижению производительности, так как системе надо осуществить необходимые преобразования (выделение памяти и копирование). В unity есть аналогичный класс UnityEngine.Object от которого наследуются все объекты.

Все классы по умолчанию могут наследоваться, но есть ряд ограничений:

- Не поддерживается множественное наследование – класс может наследоваться только от одного класса. Хотя проблема множественного наследования решается с помощью концепции интерфейсов;
- При создании производного класса надо учитывать тип доступа к базовому классу – тип доступа к производному классу должен быть таким же, как у базового класса, или более строгим. Если базовый класс имеет тип доступа **internal**, то производный класс может иметь тип доступа **internal** или **private**, но не **public**;
- Если класс объявлен с модификатором **sealed**, от него нельзя наследовать и создавать производные классы.

В Unity если класс не должен выступать в роли базового класса, то его нужно пометить модификатором **sealed**. Тогда мы немного ускорим обращение к членам этого класса.

У нас в игре будут различные главные персонажи и у них у всех будет общая логика, которая будет в базовом классе. Класс Player будет базовым.

```
namespace Geekbrains
{
    public class Player
    {
        public float Speed = 3.0f;

        public Player(float speed)
        {
            Speed = speed;
        }
    }
}
```

Создадим класс его наследника PlayerBall.

```
namespace Geekbrains
{
    public sealed class PlayerBall : Player
```

```
    {
        public PlayerBall(float speed) : base(speed)
        {

        }
    }
}
```

Наследоваться могут поля, методы, свойства и функционал класса. Конструкторы и значение полей и свойств не наследуется.

Как было ранее сказано, конструкторы не передаются производному классу при наследовании. И если в базовом классе не определен конструктор по умолчанию без параметров (а только конструкторы с параметрами), то в производном классе мы обязательно должны вызвать один из этих конструкторов через ключевое слово **base**.

Мы создали новый класс PlayerBall, который наследует публичные (**public**) и защищенные (**protected**) члены класса Player.

Теперь мы можем добавлять в новый класс собственные поля, методы и свойства, а также пользоваться неприватными полями, методами и свойствами базового класса.

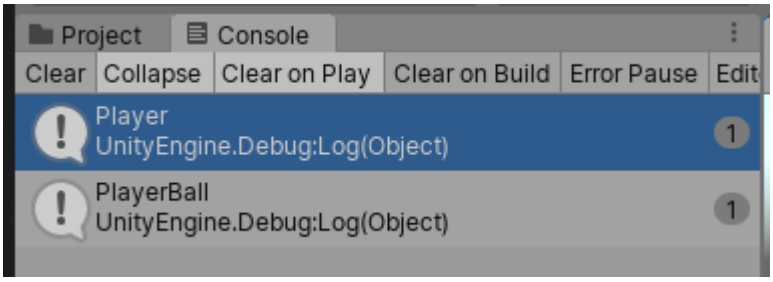
Немного перепишем код для примера и создадим класс PlayerBall.

```
namespace Geekbrains
{
    public class Player
    {
        public Player()
        {
            Debug.Log("Player");
        }
    }
}
```

```
namespace Geekbrains
{
    public sealed class PlayerBall : Player
    {
        public PlayerBall()
        {
            Debug.Log("PlayerBall");
        }
    }
}
```

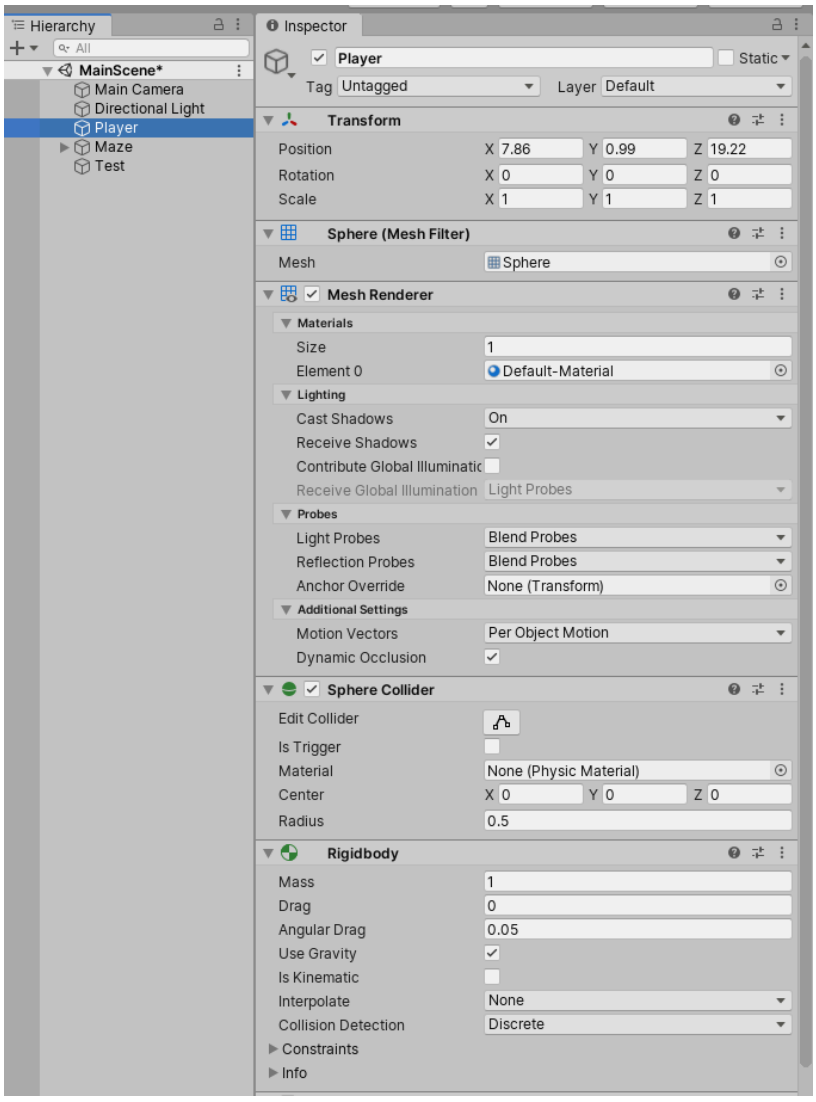
```
namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            Player player = new PlayerBall();
        }
    }
}
```

Обратите внимание, как происходят вызовы конструкторов объектов. Важно понять, что для построения нашего объекта должен быть сконструирован объект базового класса и все объекты в иерархии наследования.



В отличие от классов, структуры не поддерживают наследование. Структура не может наследовать от другой структуры или класса и не может быть базовой для класса. По этой причине члены структуры не могут объявляться как **protected**. Структуры могут реализовывать интерфейсы именно так, как это делают классы.

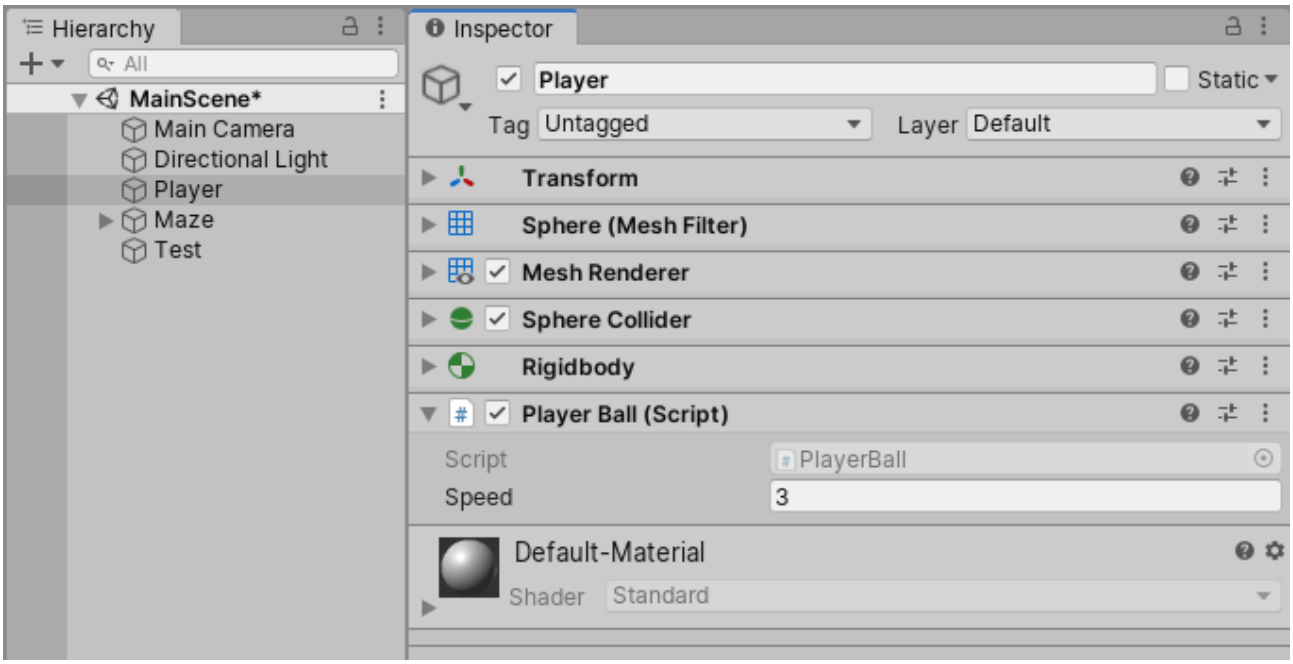
Ранее наследование вы могли увидеть, когда писали классы, которые наследовались от MonoBehaviour. Эти классы не рекомендуется создавать с помощью оператора new. А перетаскивать как компонент на GameObject.



Все компоненты, которые вы видите в inspector, это классы. Когда вы перетаскиваете компоненты на GameObject, у них вызовется конструктор и они создадутся в управляемой куче. Не рекомендуется переопределять конструкторы у наследников класса MonoBehaviour, а инициализацию объекта производить в методах Awake, Start, OnEnable и OnValidate.

Отнаследуем класс Player от класса MonoBehaviour для того чтобы можно было его наследников добавить на GO Player.

```
using UnityEngine;

namespace Geekbrains
{
    public class Player : MonoBehaviour
    {
        public float Speed = 3.0f;
    }
}
```



Добавим в класс Player метод Move, для перемещения персонажа. Перемещение будет реализовано через физический компонент. Для этого нам необходимо получить ссылку на конкретный Rigidbody. Воспользуемся методом GetComponent, Данный метод найдет экземпляр класса, который мы укажем в типопезированных скобках, на том GO который мы укажем.

```
using UnityEngine;

namespace Geekbrains
{
    public class Player : MonoBehaviour
    {
        public float Speed = 3.0f;
        private Rigidbody _rigidbody;

        private void Start()
        {
            _rigidbody = GetComponent<Rigidbody>();
        }

        protected void Move()
        {
            float moveHorizontal = Input.GetAxis("Horizontal");
            float moveVertical = Input.GetAxis("Vertical");

            Vector3 movement = new Vector3(moveHorizontal, 0.0f, moveVertical);

            _rigidbody.AddForce(movement * Speed);
        }
    }
}
```

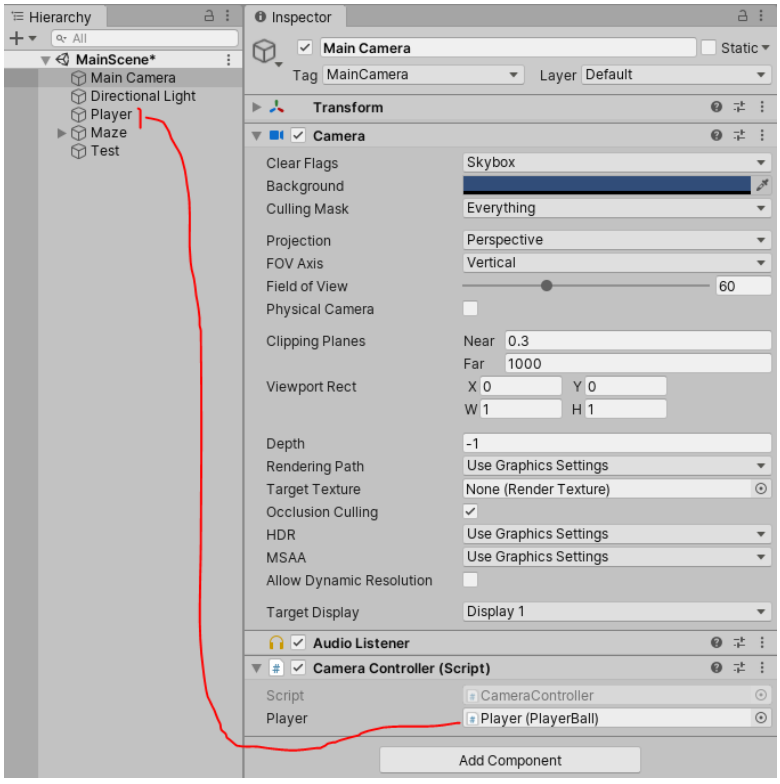
Метод Move вызовем в методе FixedUpdate класса PlayerBall. Тепер нашим персонажем можно управлять.

```
namespace Geekbrains
{
    public sealed class PlayerBall : Player
    {
```

```
        private void FixedUpdate()
        {
            Move();
        }
    }
}
```

Теперь нам нужно чтобы камера следила за персонажем. Создадим класс и добавим на главную камеру

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class CameraController : MonoBehaviour
    {
        public Player Player;
        private Vector3 _offset;

        private void Start ()
        {
            _offset = transform.position - Player.transform.position;
        }

        private void LateUpdate ()
        {
            transform.position = Player.transform.position + _offset;
        }
    }
}
```



Полиморфизм

Полиморфизм – это свойство, позволяющее одному интерфейсу получать доступ к общему классу действий. Здесь «интерфейс» – это способ взаимодействия. Про интерфейсы, как технологию программирования, мы поговорим на следующем уроке.

Пример полиморфизма мы видели ранее, когда вызвали в классе наследнике PlayerBall метод Move базового класса Player.

Виртуальный метод

Виртуальным называется такой метод, который объявляется как **virtual** в базовом классе. Он отличается тем, что может быть переопределен в одном или нескольких производных классах. Для переопределения используется **override**.

При определении класса-наследника и наследовании методов базового класса мы можем выбрать одну из следующих стратегий:

1. Обычное наследование всех членов базового класса в классе-наследнике;
2. Переопределение членов базового класса в классе-наследнике;
3. Скрытие членов базового класса в классе-наследнике.

```
namespace Geekbrains
{
    public class InteractiveObject
    {
        public virtual string DisplayFirstWay()
        {
            return $"I am a {nameof(InteractiveObject)} class method";
        }

        public virtual string DisplaySecondWay()
        {
            return $"I am a {nameof(InteractiveObject)} class method";
        }

        public virtual string DisplayThirdWay()
        {
            return $"I am a {nameof(InteractiveObject)} class method";
        }
    }
}
```

```
namespace Geekbrains
{
    public sealed class GoodBonus : InteractiveObject
    {
        public override string DisplaySecondWay()
        {
            return $"I am a {nameof(GoodBonus)} class method";
        }

        public new string DisplayThirdWay()
        {
```

```
            return $"I am a {nameof(GoodBonus)} class method";
        }
    }
}
```

```
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            InteractiveObject interactiveObject = new InteractiveObject();
            Debug.Log(interactiveObject.DisplayFirstWay());
            Debug.Log(interactiveObject.DisplaySecondWay());
            Debug.Log(interactiveObject.DisplayThirdWay());

            Debug.Log("++++++++++++++++++++++++++");

            InteractiveObject interactiveObject1 = new GoodBonus();
            Debug.Log(interactiveObject1.DisplayFirstWay());
            Debug.Log(interactiveObject1.DisplaySecondWay());
            Debug.Log(interactiveObject1.DisplayThirdWay());

            Debug.Log("++++++++++++++++++++++++++");

            GoodBonus interactiveObject2 = new GoodBonus();
            Debug.Log(interactiveObject2.DisplayFirstWay());
            Debug.Log(interactiveObject2.DisplaySecondWay());
            Debug.Log(interactiveObject2.DisplayThirdWay());
        }
    }
}
```

InteractiveObject будет выступать в роли базового класса для всех объектов с которыми может взаимодействовать персонаж. Добавим виртуальный метод для взаимодействия с объектом.

```
namespace Geekbrains
{
    public class InteractiveObject
    {
        protected virtual void Interaction()
        {

        }
    }
}
```

```
namespace Geekbrains
{
    public sealed class GoodBonus : InteractiveObject
    {
        protected override void Interaction()
        {
            base.Interaction();
            // Add bonus
        }
    }
}
```

Хорошим тоным является дополнять функционал базового класса, а не переписывать его. Поэтому при переопределении метода, вначале вызывается функционал базового класса через ключевое слово base. Потом пишется свой функционал.

Событийные методы Unity такие как Awake, Start Update и т.д. тоже можно делать виртуальными.

Оставлять пустой виртуальный метод считается дурным тоном. Потому что не факт что дочернии классы реализуют этот метод. Если мы хотим заставить дочернии классы реализовать метод базавого класса, то можно воспользоваться абстрактном типом данных.

Abstract

Если мы помечаем класс как абстрактный, то нельзя создавать экземпляр данного класса, а это значит что данный класс будет выступать только в роли базового класса. И второе, данный класс может иметь абстрактные методы, свойства и индексаторы. Индексаторы будут рассказываться в последующих лекциях.

```
namespace Geekbrains
{
    public abstract class InteractiveObject
    {
        protected abstract void Interaction();
    }
}
```

```
namespace Geekbrains
{
    public sealed class GoodBonus : InteractiveObject
    {
        protected override void Interaction()
        {
            // Add bonus
        }
    }
}
```

Абстрактные методы не могут быть приватными (должны быть либо **public**, либо **protected**). По сути, абстрактный метод – это виртуальный метод, только без определения поведения. Подразумевается, что поведение абстрактного метода будет реализовано в классах-наследниках. Абстрактный метод может быть описан только в абстрактном классе.

Зачем определять класс, экземпляр которого нельзя создать непосредственно? Базовые классы (абстрактные или нет) очень полезны: они содержат общие данные и общую функциональность для унаследованных типов. Используя эту форму абстракции, можно также моделировать общую «идею», а не обязательно конкретную сущность. И хотя непосредственно создать экземпляр абстрактного класса нельзя, он все же присутствует в памяти, когда создан экземпляр его производного класса. Таким образом, совершенно нормально (и принято) для абстрактных классов определять любое количество конструкторов, вызываемых опосредованно при размещении в памяти экземпляров производных классов.

ToString()

На самом деле, мы с вами уже несколько раз переопределяли (точнее, скрывали) виртуальный метод **ToString**. Все классы в **.Net Framework** наследуются от базового класса **System.Object**, в котором определены несколько виртуальных методов: **ToString()**, **Equals()**, **GetHashCode()**.

Это сделано специально, чтобы наследники (все классы) могли их переопределить и сделать их поведение более естественным для их объекта. Иначе вызывается поведение, заданное в **System.Object** – но оно не может знать о реализации вашего объекта и ведет себя довольно примитивно. Например, не переопределенный (принадлежащий **System.Object**) метод **ToString** выводит информацию о том, к какому классу и пространству имен принадлежит созданный вами класс.

Чтобы правильно переопределить поведение **ToString**, мы должны добавить слово **override** перед описанием данного метода в нашем классе. Без **override** мы скрываем базовое поведение и не даем будущим потомкам нашего класса использовать поведение, заложенное в базовом классе:

Программист может использовать уже готовые классы, разрабатывать собственные или изменять другие классы, используя наследование и иные механизмы.

IS и AS

IS проверяет совместимость объекта с заданным типом, AS — проверяет и, при возможности, преобразует объект. В следующем коде определяется, является ли объект экземпляром типа GoodBonus, производного от InteractiveObject:

```
using UnityEngine;

namespace Geekbrains
{
   public class Test : MonoBehaviour
   {
       private void Start()
       {
           InteractiveObject interactiveObject = new GoodBonus();

           Debug.Log(interactiveObject is GoodBonus);
       }
   }
}
```

Оператор **as** используется для преобразования типов между совместимыми ссылочными типами или для типа, допускающего значение **NULL**. Он подобен оператору приведения **(int)x**. Если преобразование невозможно, **as** возвращает **null** вместо вызова исключения.

```
using UnityEngine;

namespace Geekbrains
{
   public class Test : MonoBehaviour
   {
       private void Start()
       {
           InteractiveObject interactiveObject = new InteractiveObject();
```

```
            Debug.Log(interactiveObject as GoodBonus);
        }
    }
}
```

Начиная с C# 7 добавилась функциональность pattern matching

pattern matching

```
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            InteractiveObject interactiveObject= new InteractiveObject();

            if (interactiveObject is GoodBonus bonus)
            {
                bonus.DisplayFirstWay();
            }
        }
    }
}
```

Кроме выражения if pattern matching может применяться в конструкции switch:

```
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            InteractiveObject interactiveObject = new InteractiveObject();

            Example(interactiveObject);

            void Example(InteractiveObject value)
            {
                switch (value)
                {
                    case GoodBonus goodBonus when enabled:
                        break;
                    case null:
                        break;
                    default:
                        break;
                }
            }
        }
    }
}
```

С помощью выражения **when** можно вводить дополнительные условия в конструкцию case.

По причине плохой перегрузки оператора сравнения Unity объектов, не рекомендуется часто проверять на null эти объекты, а проверка на null через элвис оператор не работает.

Локальные методы

В примере выше используются локальный метод. Он будет доступен только в контексте того метода в котором он был создан. 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.

Значение null и Nullable-типы

Бывают случаи, когда программистам удобно, чтобы объекты значимых типов данных имели значение **null**, то есть были бы не определены. Для этого надо использовать знак вопроса (**?**) после типа значений.

```
int? test = null;
bool? isEnabled = null;
```

Запись **?** является упрощенной формой использования структуры **System.Nullable<T>**. Это обобщенная структура (рассмотрим обобщения в последующих уроках).

```
int? test = 5;
bool? isEnabled = null;
double? pi = 3.14;

Nullable<int> test1 = 5;
Nullable<bool> isEnabled1 = null;
Nullable<double> pi1 = 3.14;
```

Чтобы получать значение объекта, необходимо обратиться к свойству Value:

```
int? b = 1;
if (b.HasValue) // Прежде чем получить значение объекта, необходимо проверить, хранит
ли объект какое-либо значение
{
    Debug.Log($"Значение b = {b.Value}");
}
```

Оператор **??** называется оператором null-объединения. Он применяется для установки значений по умолчанию для типов значений и ссылочных типов, которые допускают значение **null**. Оператор **??** возвращает левый операнд, если он не равен **null** – иначе возвращается правый операнд (в этом случае левый операнд должен принимать null):

```
int? x = null;
int y = x ?? 2;  // Равно 2, так как x равен null

int? a = 5;
int b = a ?? 10; // Равно 5, так как a не равен null
```

С версии C# 6.0 в языке появился оператор условного **null** (**Null-Conditional Operator),** или элвис-оператор. Он позволяет упростить проверку на значение **null** в условных конструкциях:

```
using UnityEngine;

namespace Geekbrains
```

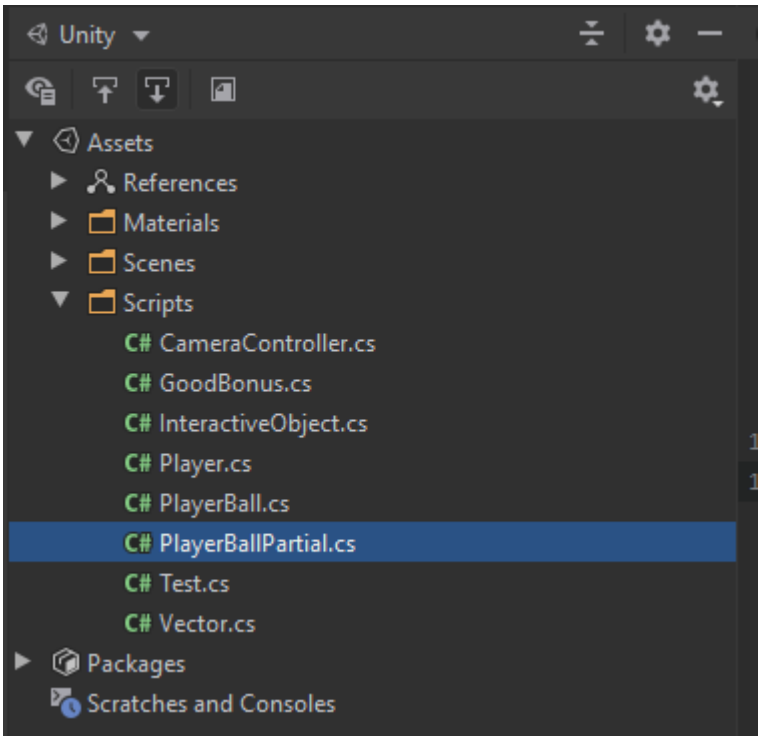
```
{
    public sealed class Example
    {
        public TestClass Object;
    }

    public sealed class TestClass
    {
        public string Name;
    }

    public class Test : MonoBehaviour
    {
        private void Start()
        {
            Example example = new Example();

            Debug.Log(example?.Object?.Name ?? "");
        }
    }
}
```

Partial

Классы могут быть частичными. То есть мы можем иметь несколько файлов с определением одного и того же класса, и при компиляции все эти определения будут скомпилированы в одно.

```
namespace Geekbrains
{
    public sealed partial class PlayerBall : Player
    {
        private void FixedUpdate()
        {
            Move();
            Jump();
        }
    }
}
```

```
namespace Geekbrains
{
    public sealed partial class PlayerBall
    {
        private void Jump()
        {

        }
    }
}
```



Частичные классы могут содержать частичные методы. При этом частичные методы не могут иметь модификаторов доступа - по умолчанию они все считаются приватными. Также частичные методы не могут иметь таких модификаторов как virtual, abstract, override, new, sealed. Хотя допустимы статические частичные методы. Кроме того, частичные методы не могут возвращать значения, то есть они всегда имеют тип void. И также они не могут иметь out-параметров. Поскольку частичные методы всегда приватные, то мы не сможем их вызвать напрямую в программе вне классов

Советы

Правила для названий классов и методов

Обычно классы представляют объекты, а методы – действия. Поэтому для наименования классов используйте существительные – Cat, Machine, Girl. При создании методов используйте глаголы в следующей нотации – MoveLeft, ShowMessage. Так вам и другим программистам будет проще ориентироваться в коде.

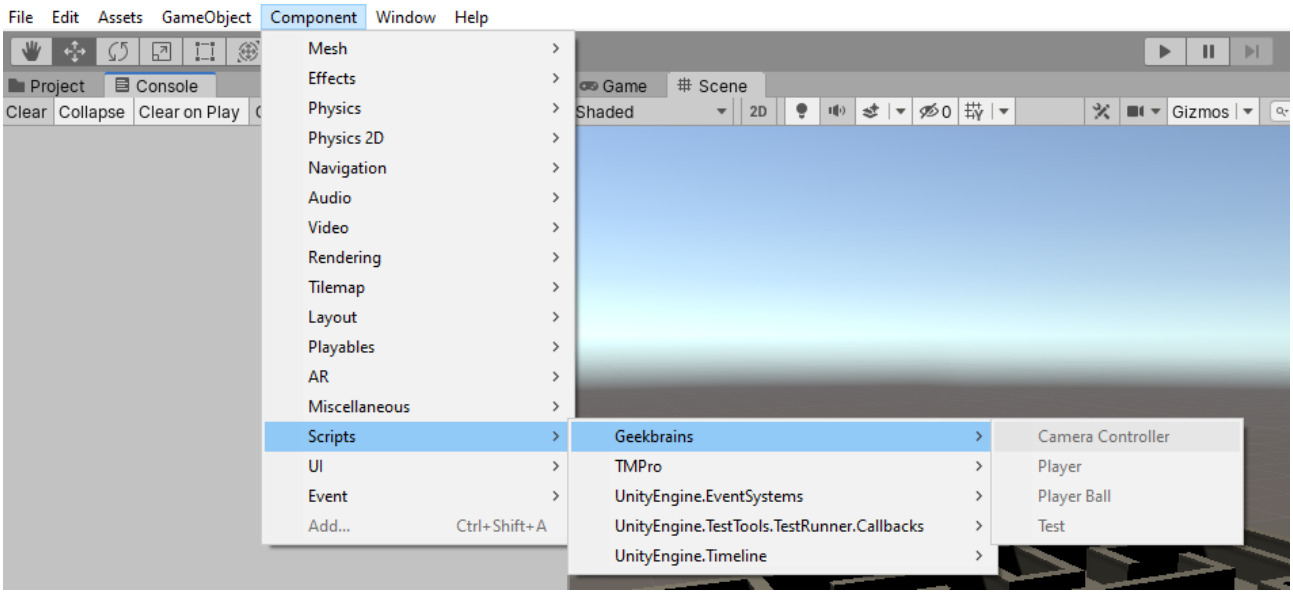
Пространства имен

Оборачивайте весь ваш код в пространство имен, это вам поможет избежать конфликтов при импорте других проектов.

```
using UnityEngine;

public class Example
{
    public int Test;
}

namespace Geekbrains
{
    public class Example
    {
```

```
        public string Test;
    }

    namespace MyNamespace.Geekbrains
    {
        public class Example
        {
            public float Test;
        }
    }

    public class Test : MonoBehaviour
    {
        private void Start()
        {
            Geekbrains.MyNamespace.Geekbrains.Example example1 = new
MyNamespace.Geekbrains.Example();

            Geekbrains.Example example2 = new Example();

            global::Example example3 = new global::Example();
        }
    }
}
```

А в юнити ваши классы будут сгруппированы



Практическое задание

1. Спроектировать и создать уровень
2. Создать основные классы и обернуть их в пространство имен. Подготовить диаграмму классов
3. *Разработать игрока с альтернативным управлением

Дополнительные материалы

1. Полный список перегружаемых операторов: документация msdn;
2. https://blogs.unity3d.com/ru/2014/05/16/custom-operator-should-we-keep-it/
3. https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/structs;
4. https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/nullable-types/using-nullable-types.

Используемая литература

При создании 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.