



Урок 1

Принципы ООП в Unity. Часть 1

Пространство имен. Классы, поля, методы, свойства. Конструкторы. Наследование, полиморфизм. Виртуальные методы. Перегрузка методов.

[Введение](#)

[Компонентный подход](#)

[Объектно-ориентированный подход](#)

[Приступаем к написанию 3D-шутера](#)

[Подготовка проекта](#)

[Базовый класс всех объектов – BaseObjectScene](#)

[Базовый класс всего оружия – Weapons](#)

[Базовый класс всех контроллеров – BaseControllery](#)

[Контроллер, отвечающий за работу фонарика](#)

[Контроллер, отвечающий за горячие клавиши](#)

[Главная точка входа в программу – класс Main](#)

[Итоги](#)

[Домашнее задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Введение

Создание сложных программных приложений связано с разделением на этапы жизненного цикла:

1. Анализ предметной области и создание ТЗ (взаимодействие с заказчиком);
2. Проектирование структуры программы;
3. Кодирование (набор программного кода согласно проектной документации);
4. Тестирование и отладка;
5. Внедрение программы;
6. Сопровождение программы;
7. Утилизация.

При проектировании игр, как и любой другой программы, существует множество подходов, которые зачастую комбинируются. Рассмотрим несколько примеров проектирования игр в Unity3D.

Компонентный подход

Суть компонентно-ориентированного программирования (КОП) заключается в том, что к существующему объекту добавляются компоненты, которые отвечают за его поведение и взаимодействие с внешним миром. Например:

1. Воин ближнего боя. Его компоненты:
 - a. Transform – определяет позицию, поворот и масштаб;
 - b. MiddleAttack – компонент для атаки в ближнем бою;
 - c. Moving – компонент перемещения;
 - d. MeshCollider – компонент обработки столкновений.
2. Воин дальнего боя. Его компоненты:
 - a. Transform ;
 - b. DistanceAttack – компонент для дальней атаки;
 - c. Moving;
 - d. MeshCollider.
3. Атакующее сооружение, его компоненты:
 - a. Transform;
 - b. DistanceAttack;
 - c. MeshCollider.

КОП упрощает повторное использование написанного кода – один компонент применим в разных объектах. Из комбинаций уже существующих компонентов можно собрать новый тип объекта.

Объектно-ориентированный подход

При проектировании игры мы будем использовать все достоинства объектно-ориентированного программирования (ООП). Разделим объекты на группы, и у каждой будет свой базовый класс. Прибегнем к шаблону проектирования MVC: будем использовать контроллеры для управления данными. Введем в проект поведенческие классы. Составим предварительную диаграмму классов:

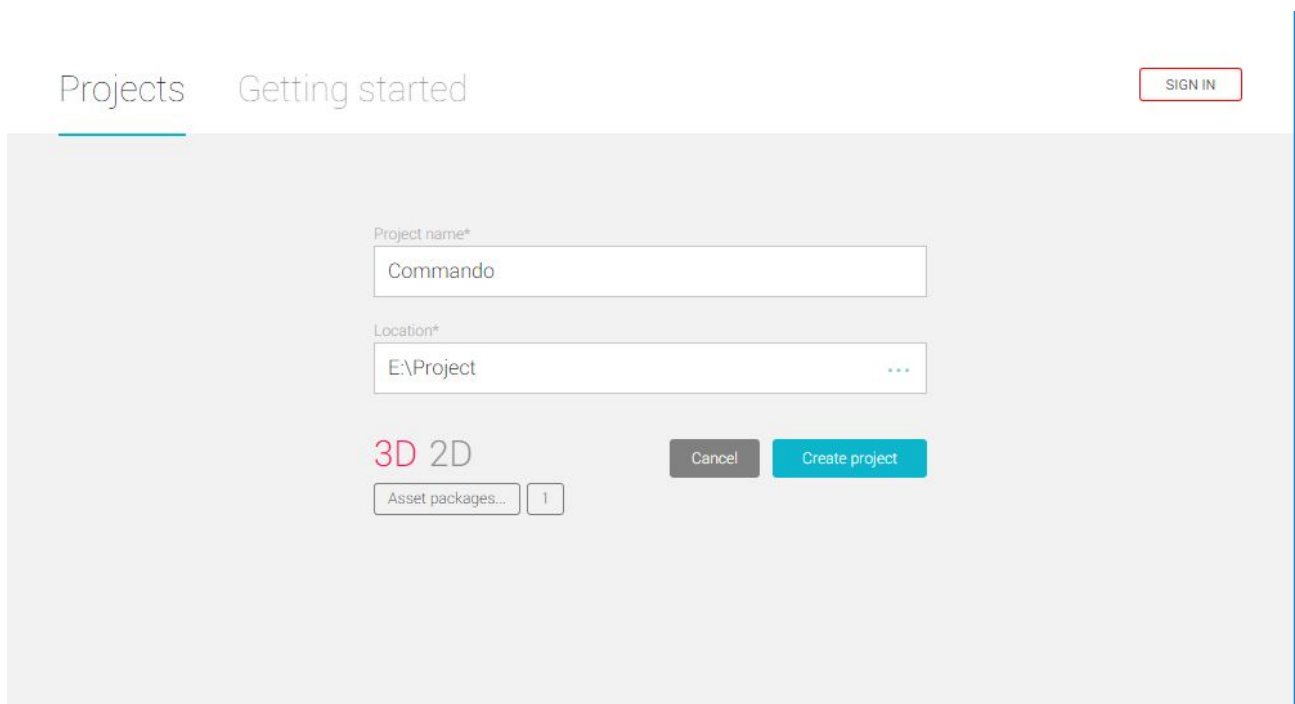
игры. По окончании вебинара вы получите скелет игры, на который можно установить любой функционал.

Приступаем к написанию 3D-шутера

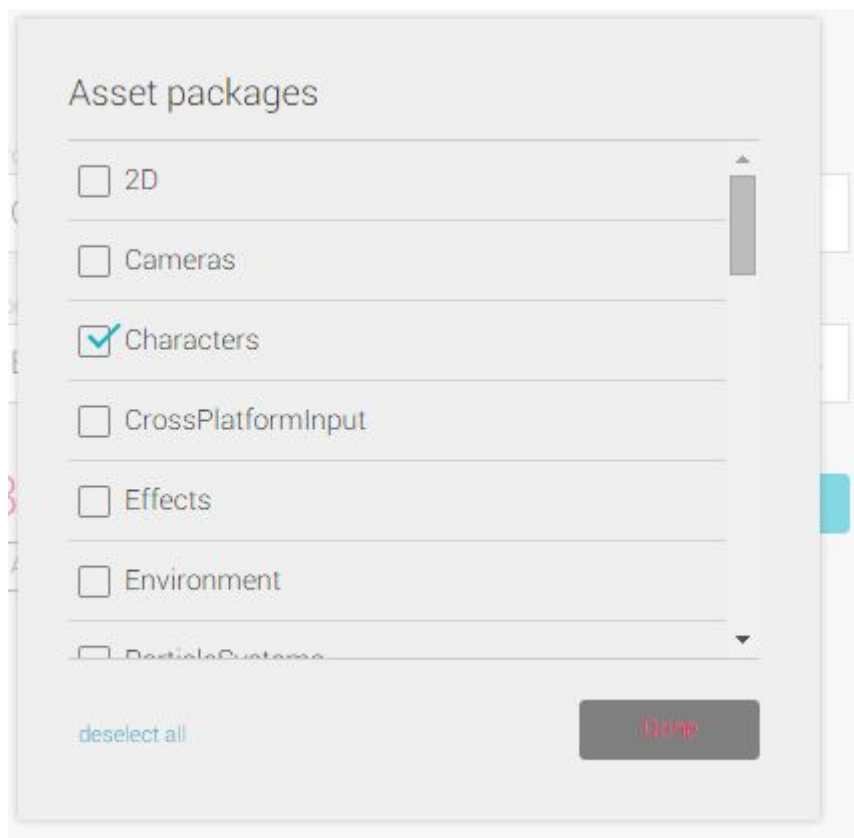
Подготовка проекта

Скачиваем и устанавливаем [Unity3d](#) последней версии. Желательно еще установить [ReSharper](#) – он упрощает процесс программирования. Эта настройка – платная, но доступна пробная версия, которая рассчитана на 30 дней (хватит на весь курс обучения). Для отладки программы стоит скачать [этот](#) инструмент.

После установки всех программ и компонентов запускаем Unity:



Задаем имя проекту и путь, где он будет создан. Указываем, что это 3D-проект и загружаем дополнительные ассеты. Нам понадобится только Characters.



В качестве контроллера перемещения главного персонажа будем использовать стандартный контроллер из ассета Characters – для демонстрации он подходит идеально. Когда вы будете создавать собственные коммерческие проекты, придется разработать свой контроллер управления персонажем. Наилучшее решение – дополнить и разгрузить стандартный контроллер под нужды программы.

Базовый класс всех объектов – BaseObjectScene

После проектирования приступаем к разработке игры. Для оптимизации проекта мы создали класс **BaseObjectScene**, который будет кэшировать данные и выполнять общие действия для всех объектов на сцене. Теперь они будут наследоваться не от **MonoBehaviour**, а от **BaseObjectScene**.

```
namespace Geekbrains
{
    /// <summary>
    /// Базовый класс всех объектов на сцене
    /// </summary>
    public abstract class BaseObjectScene : MonoBehaviour
    {
    }
}
```

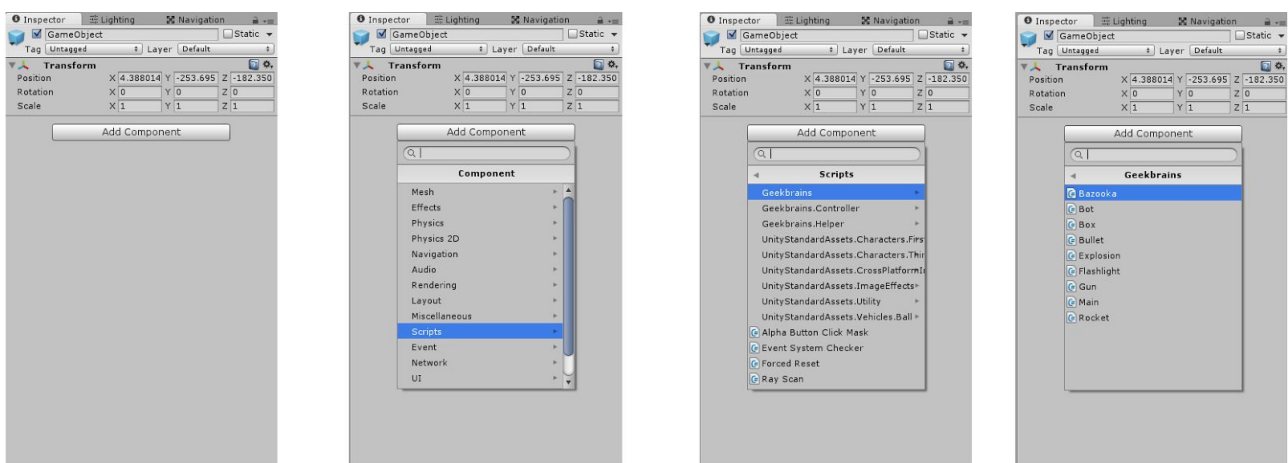
BaseObjectScene наследуется от **MonoBehaviour**. Значит, все наследники первого воспримут функции от второго. К классу добавлен модификатор **abstract**, а значит:

1. Создавать экземпляры абстрактного класса нельзя;
2. Абстрактные классы могут содержать абстрактные методы и методы доступа. Их реализация обязательна в классах-наследниках.

Разберем строчку «namespace Geekbrains». Это значит, что класс BaseObjectScene находится в пространстве имен Geekbrains. Имена и подимена можно придумать любые, например:

- 1) Roman;
- 2) Geekbrains;
- 3) Geekbrains.Interface;
- 4) Geekbrains.Helper;
- 5) Geekbrains.Controller.

Добавление класса в пространство имен дает возможность в одном проекте добавлять классы с одинаковыми именами – но при условии, что они будут обернуты в разные пространства имен. Еще получаем удобную навигацию при добавлении компонента:



Комментарий к классу – конструкция, которая делает чтение кода более удобным и дает подсказку программистам, которые будут использовать ваши классы и методы.

Добавим в класс поля, которые будут кэшироваться и использоваться всеми наследниками.

```

namespace Geekbrains
{
    /// <summary>
    /// Базовый класс всех объектов на сцене
    /// </summary>
    public abstract class BaseObjectScene : MonoBehaviour
    {
        protected int _layer;
        protected Color _color;
        protected Material _material;
        protected Transform _myTransform;
        protected Vector3 _position;
        protected Quaternion _rotation;
        protected Vector3 _scale;
        protected GameObject _instanceObject;
        protected Rigidbody _rigidbody;
        protected string _name;
        protected bool _isVisible;
    }
}

```

Модификатор доступа **protected** делает поля доступными всем наследникам, но только внутри класса. Закэшируем эти свойства объекта в функции, унаследованной от **MonoBehaviour Awake()**, и пометим ее модификатором **virtual** для переопределения у наследников.

```

#region UnityFunction
protected virtual void Awake()
{
    _instanceObject = gameObject;
    _name = _instanceObject.name;
    if (GetComponent<Renderer>())
    {
        _material = GetComponent<Renderer>().material;
    }
    _rigidbody = _instanceObject.GetComponent<Rigidbody>();
    _myTransform = _instanceObject.transform;
}
#endregion

```

В этом коде мы получили ссылки на компоненты, которые будут часто использоваться всеми объектами на сцене — закэшировали компоненты объекта. Конструкция **#region ПроизвольноеИмяРегиона #endregion** служит только для читабельности кода. Программист группирует код и оборачивает в регионы. Можно выделить в группы стандартные функции Unity, публичные и приватные функции, свойства и так далее.

Напишем свойства для компонентов, чтобы получить возможность доступа к ним из других классов.

```

#region Property
    /// <summary>
    /// Имя объекта
    /// </summary>
    public string Name
    {
        get { return _name; }
        set
        {
            _name = value;
            InstanceObject.name = _name;
        }
    }
    /// <summary>
    /// Слой объекта
    /// </summary>
    public int Layers
    {
        get { return _layer; }
        set
        {
            _layer = value;

            if (_instanceObject != null)
            {
                _instanceObject.layer = _layer;
            }
            if (_instanceObject != null)
            {
                AskLayer(GetTransform, value);
            }
        }
    }
    /// <summary>
    /// Цвет материала объекта
    /// </summary>
    public Color Color
    {
        get { return _color; }
        set
        {
            _color = value;
            if (_material != null)
            {
                _material.color = _color;
            }
            AskColor(GetTransform, _color);
        }
    }
    public Material GetMaterial
    {
        get { return _material; }
    }

```



```

    }

    /// <summary>
    /// Позиция объекта
    /// </summary>
    public Vector3 Position
    {
        get
        {
            if (InstanceObject != null)
            {
                _position = GetTransform.position;
            }
            return _position;
        }
        set
        {
            _position = value;
            if (InstanceObject != null)
            {
                GetTransform.position = _position;
            }
        }
    }

    /// <summary>
    /// Размер объекта
    /// </summary>
    public Vector3 Scale
    {
        get
        {
            if (InstanceObject != null)
            {
                _scale = GetTransform.localScale;
            }
            return _scale;
        }
        set
        {
            _scale = value;
            if (InstanceObject != null)
            {
                GetTransform.localScale = _scale;
            }
        }
    }

    /// <summary>
    /// Поворот объекта
    /// </summary>
    public Quaternion Rotation
    {
        get
        {

```

```

        if (InstanceObject != null)
        {
            _rotation = GetTransform.rotation;
        }

        return _rotation;
    }
    set
    {
        _rotation = value;
        if (InstanceObject != null)
        {
            GetTransform.rotation = _rotation;
        }
    }
}

/// <summary>
/// Получить физическое свойство объекта
/// </summary>
public Rigidbody GetRigidbody
{
    get { return _rigidbody; }
}

/// <summary>
/// Ссылка на gameObject
/// </summary>
public GameObject InstanceObject
{
    get { return _instanceObject; }
}

/// <summary>
/// Скрывает/показывает объект
/// </summary>
public bool IsVisible
{
    get { return _isVisible; }
    set
    {
        _isVisible = value;
        if(_instanceObject.GetComponent<MeshRenderer>())
            _instanceObject.GetComponent<MeshRenderer>().enabled =
_isVisible;

        if (_instanceObject.GetComponent<SkinnedMeshRenderer>())
            _instanceObject.GetComponent<SkinnedMeshRenderer>().enabled =
_isVisible;
    }
}

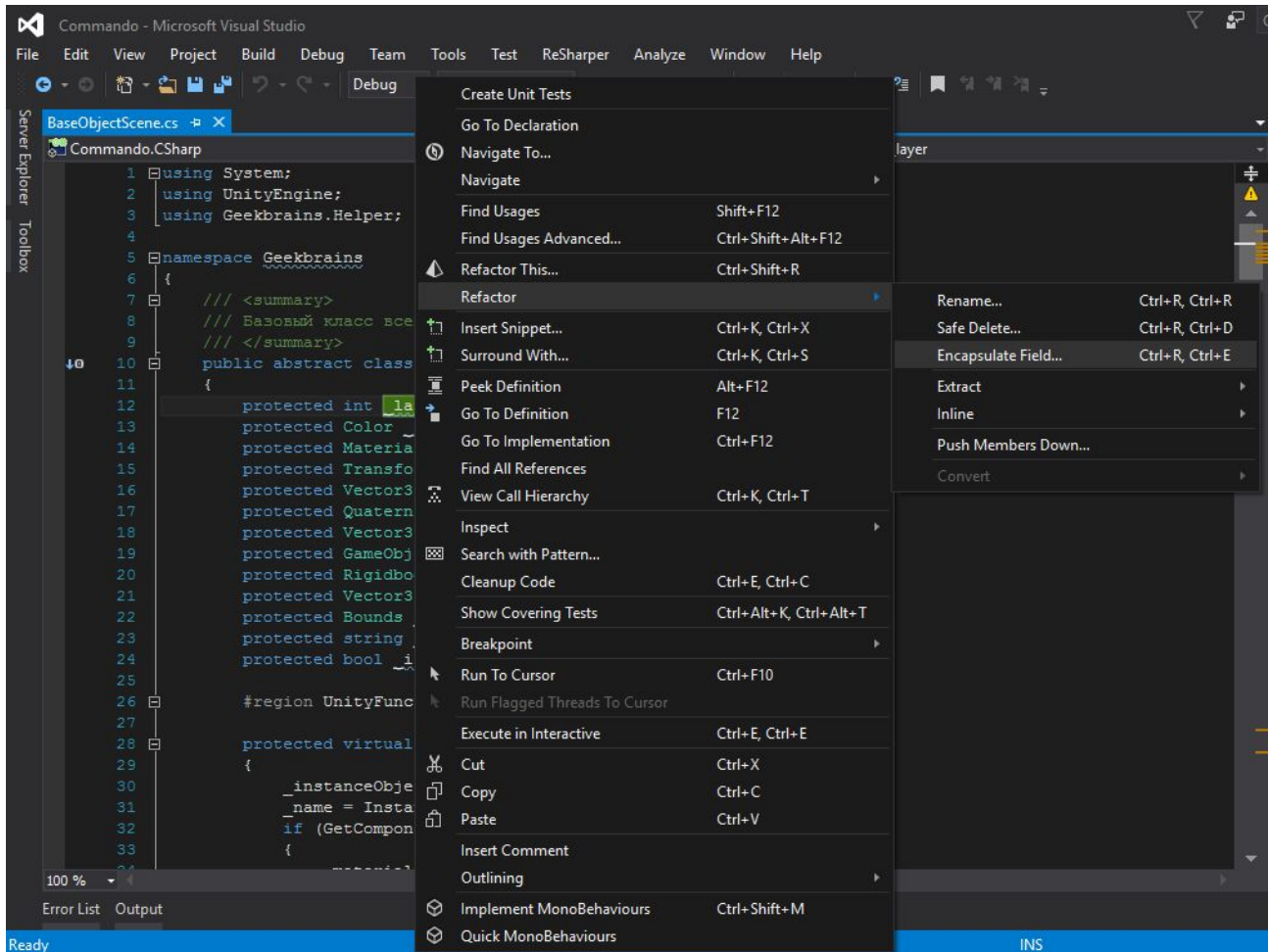
/// <summary>
/// Получить Transform объекта
/// </summary>
public Transform GetTransform
{
    get { return _myTransform; }
}

```

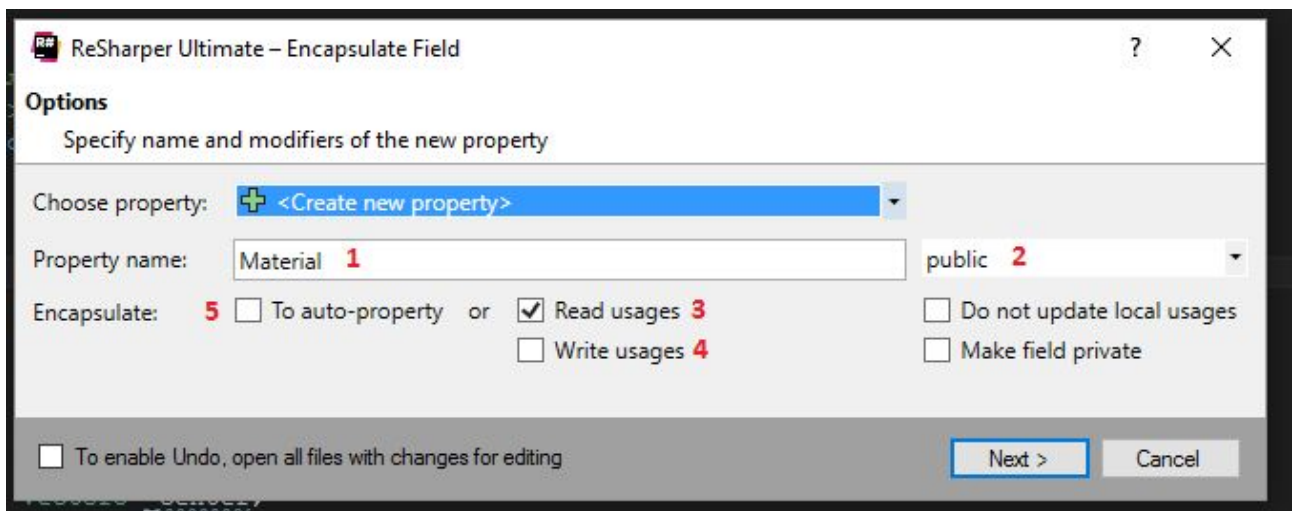
```
}

#endregion
```

Для быстрого создания свойства нажмем правой кнопкой на поле класса. В контекстном меню выберем **Refactor->Encapsulate Field**:



Появится окно «Создание свойства поля»:



1. Имя свойства;

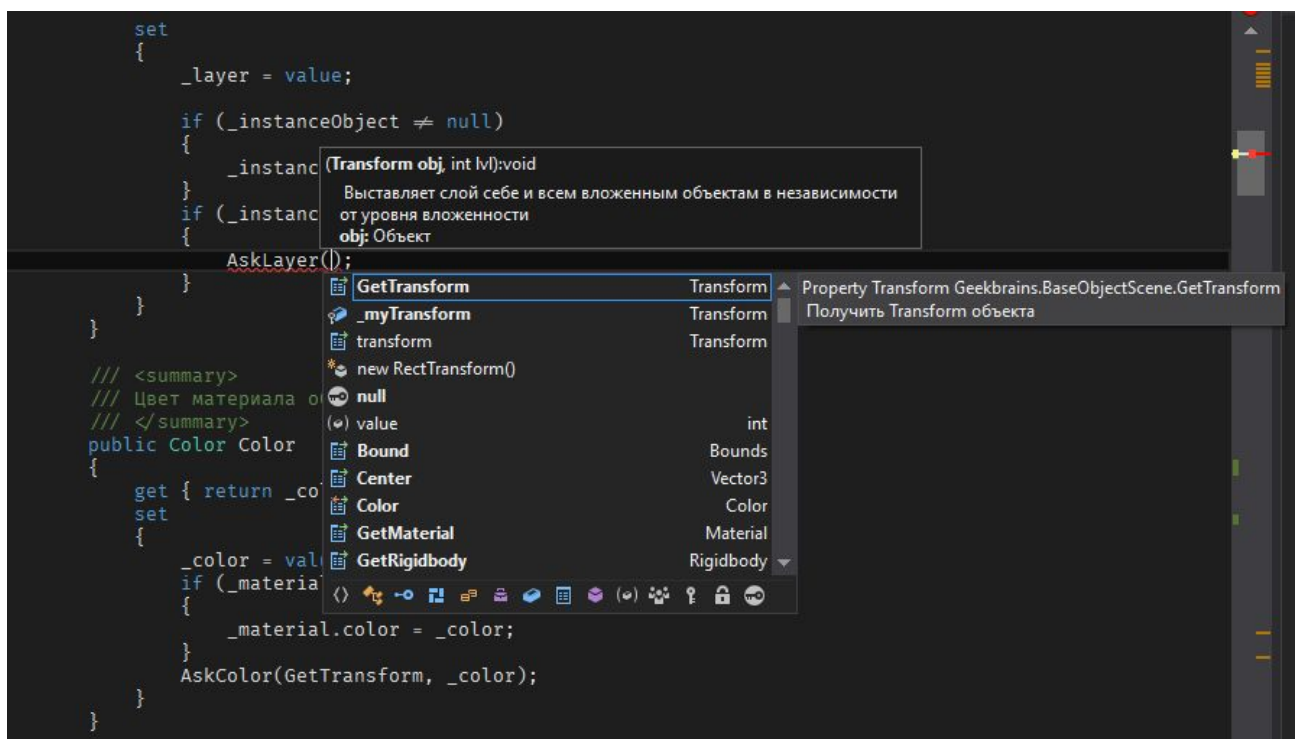
2. Модификатор доступа;
3. Получить доступ к объекту;
4. Изменить объект;
5. Автосвойство. В данном классе не нужны поля с автосвойством, и мы разберем их позже.

Рассмотрим код, приведенный выше. Помимо свойств там присутствуют 2 функции: **AskLayer(GetTransform, value)** и **AskColor(GetTransform, _color)**.

```
#region PrivateFunction
    /// <summary>
    /// Выставляет слой себе и всем вложенным объектам независимо от
уровня вложенности
    /// </summary>
    /// <param name="obj">Объект</param>
    /// <param name="lvl">Слой</param>
    private void AskLayer(Transform obj, int lvl)
    {
        obj.gameObject.layer = lvl;          // Выставляем объекту слой
        if (obj.childCount > 0)
        {
            foreach (Transform d in obj) // Проходит по всем вложенным
объектам
            {
                AskLayer(d, lvl);          // Рекурсивный вызов функции
            }
        }

        private void AskColor(Transform obj, Color color)
        {
            // Реализовать по аналогии с
AskLayer
        }
    }
#endregion
```

Этот код сопровождается комментариями, задокументирован и не требует дополнительных объяснений. Рассмотрим комментарий к функции: точно такую же конструкцию мы видели у класса **BaseObjectScene**. Она позволяет увидеть комментарий при вызове функции или создании экземпляра класса:



Базовый класс для типов объектов готов. Приступим к созданию базового класса всего оружия.

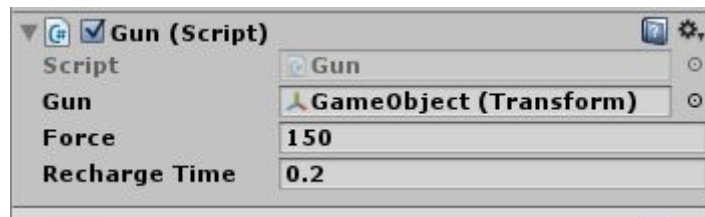
Базовый класс всего оружия – Weapons

```
using UnityEngine;
using System.Collections;
namespace Geekbrains
{
    /// <summary>
    /// Базовый класс для всех типов оружия
    /// </summary>
    public abstract class Weapons : BaseObjectScene
    {
        #region Serialize Variable
        // Позиция, из которой будут вылетать снаряды
        [SerializeField] protected Transform _gun;
        // Сила выстрела
        [SerializeField] protected float _force = 500;
        // Время задержки между выстрелами
        [SerializeField] protected float _rechargeTime = 0.2f;
        #endregion
        #region Protected Variable
        protected bool _fire = true;
        // Флаг, разрешающий выстрел
        #endregion
        #region Abstract Function
        // Функция для вызова выстрела, обязательна во всех дочерних классах
        public abstract void Fire();
        #endregion
    }
}
```

Часть полей помечаем как **SerializeField** – они будут доступны в инспекторе. А модификатор доступа **protected** означает, что эти поля будут недоступны вне класса, но унаследуются от класса **Weapons**. Создадим класс **Gun** и унаследуем его от **Weapons**.

```
using UnityEngine;
using System.Collections;
namespace Geekbrains
{
    /// <summary>
    /// Класс определяет поведение оружия «Автомат»
    /// </summary>
    public sealed class Gun : Weapons
    {
        public override void Fire()
        {
            // Функцию Fire реализуем в следующем уроке
        }
    }
}
```

От **Weapons** мы унаследовали поля, которые можем теперь проинициализировать в инспекторе:



Мы создали заготовку класса под оружие. На следующем уроке дополним его, повесим производный класс на модель оружия и научим его стрелять.

Базовый класс всех контроллеров – BaseControllery

Этот класс является базовым для всех контроллеров и хранит в себе флаг включения и выключения контроллера.

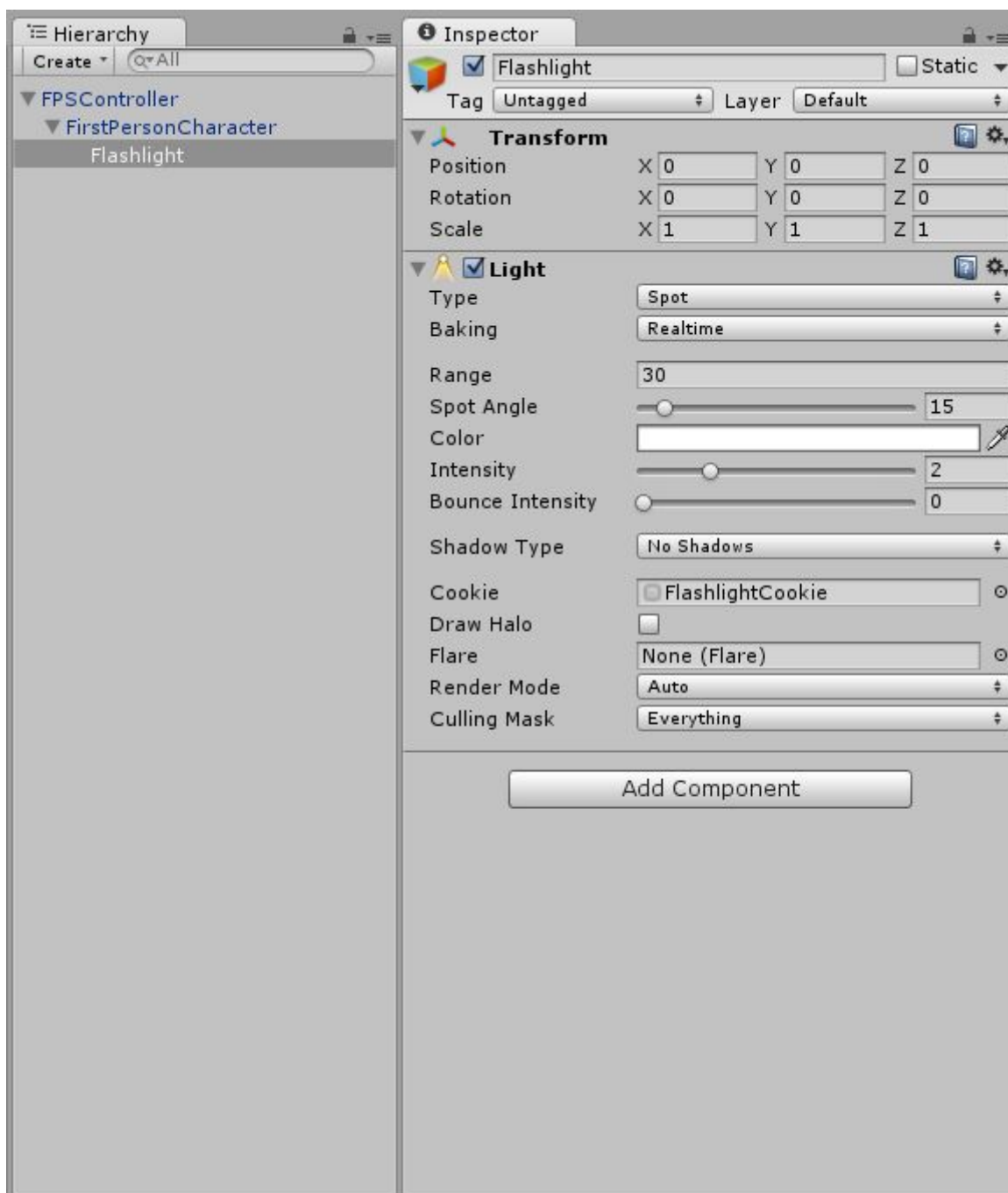
Для демонстрации работы создадим 2 контроллера. Первый – **InputController**. Он будет работать весь цикл программы и считывать ввод данных с клавиатуры. Второй – **FlashlightController**, который отвечает за работу фонарика.

Контроллер, отвечающий за работу фонарика

```
using UnityEngine;
namespace Geekbrains.Controller
{
    public sealed class FlashlightController : BaseController
    {
        private Light _light;           // Ссылка на источник света
        private void Awake()
        {
            _light = GameObject.Find("Flashlight").GetComponent<Light>();
        }
        public void Start()
        {
            SetActiveFlashlight(false); // При старте сцены выключаем фонарик
        }
        public void Update()
        {
            if (!Enabled) return;       // Если контроллер неактивен, выходим
из Update
// Здесь описываем поведение фонарика: можно добавить максимальное время его
работы, смену батареек и другое
        }
        private void SetActiveFlashlight(bool value)
        {
            _light.enabled = value;
        }
        public override void On()
        {
            if (Enabled) return;       // Если контроллер включен, повторно
не включаем
            base.On();
            SetActiveFlashlight(true);
        }
        public override void Off()
        {
            if (!Enabled) return;     // Если контроллер выключен, повторно
не выключаем
            base.Off();
            SetActiveFlashlight(false);
        }
    }
}
```

Этот класс описывает поведение фонарика: включает и выключает источник света, в него можно добавить продолжительность работы фонарика, вывод на экран заряда батареи, смену батареек и так далее.

В функции **Awake** необходимо получить ссылку на источник света. Возможны разные способы, мы поступили так: назвали источник света назвал «Flashlight», с помощью метода **Find** класса **GameObject** нашли его на сцене и взяли у него компонент **Light**:



Теперь необходимо прописать включение и выключение контроллера по нажатию клавиши.

Контроллер, отвечающий за горячие клавиши

```
using UnityEngine;
namespace Geekbrains.Controller
{
    /// <summary>
    /// Контроллер, который отвечает за горячие клавиши
    /// </summary>
    public sealed class InputController : BaseController
    {
        private bool _isActiveFlashlight = false;
        public void Update()
        {
            if (Input.GetKeyDown(KeyCode.F))
            {
                _isActiveFlashlight = !_isActiveFlashlight;
                if (_isActiveFlashlight)
                {
                    // Вызов функции On() класса FlashlightController
                }
                else
                {
                    // Вызов функции Off() класса FlashlightController
                }
            }
        }
    }
}
```

В этом классе мы ждем, когда пользователь нажмет кнопку «F» и далее действуем в зависимости от того, включен или выключен фонарик. Наша задача – получить ссылку на контроллер фонарика. Для этого воспользуемся паттерном «Одиночка». Создадим класс **Main**: в нем будут храниться ссылки на все контроллеры и хэлперы. В будущем этот класс будет структурировать игровую сцену.

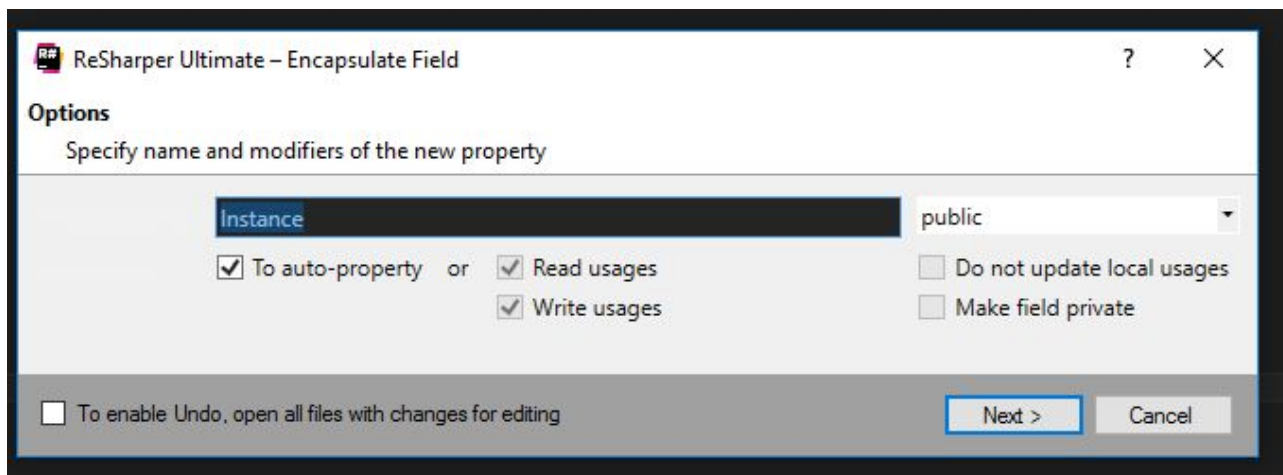
Главная точка входа в программу – класс Main

```
using UnityEngine;
using System.Collections.Generic;
using Geekbrains.Controller; // Подключаем пространство имен, в котором
находятся контроллеры
using Geekbrains.Helper; // Подключаем пространство имен, в котором
находятся хэлперы
namespace Geekbrains
{
    /// <summary>
    /// Точка входа в программу
    /// </summary>
    public sealed class Main : MonoBehaviour
    {
        private GameObject _controllersGameObject;
        private InputController _inputController;
        private FlashlightController _flashlightController;
        private static Main _instance;
        void Start()
        {
            _instance = this;
            _controllersGameObject = new GameObject {name = "Controllers"};
            _inputController =
            _controllersGameObject.AddComponent<InputController>();
            _flashlightController =
            _controllersGameObject.AddComponent<FlashlightController>();
        }
        #region Property
        /// <summary>
        /// Получить контроллер фонарика
        /// </summary>
        public FlashlightController GetFlashlightController
        {
            get { return _flashlightController; }
        }
        /// <summary>
        /// Получить контроллер ввода данных
        /// </summary>
        /// <returns></returns>
        public InputController GetInputController()
        {
            return _inputController;
        }
        #endregion
    }
}
```

Этот класс – Singleton. Создать «Одиночку» можно разными способами. Мы создали класс **Main** и унаследовали его от **MonoBehaviour** – для добавления на сцену. В этом классе создали статическое поле с таким же типом, как наш класс.

```
private static Main _instance;
```

Объект, помеченный как статический, живет на протяжении всего цикла программы – это как раз то, что нам нужно. К этому полю можно применить автосвойство. Нажимаем правой кнопкой на поле и в контекстном меню выбираем **Refactor->Encapsulate Field**. Ставим галочку напротив «**To auto-property**»:



Теперь поле класса приняло следующий вид:

```
public static Main Instance { get; private set; }
```

В функции **Start** получим ссылку на этот класс. Если бы вы разрабатывали программу не на Unity3D, то ссылку получали бы в конструкторе класса. Но в Unity3D для этого есть специальные функции: **Awake** и **Start**. Именно в них рекомендуют инициализировать данные при разработке приложений.

Пример того, **как не стоит инициализировать данные в Unity3d**:

```
public Main() // Конструктор класса Main
{
    Instance = this;
}
```

В переменной Instance находится ссылка на класс **Main**. Это значит, что мы можем из любого места в программе получить все публичные переменные и функции класса **Main**. Доступ к ним будет осуществлен так:

```
Main.Instance.(То, что мы хотим получить)
```

Теперь можем дописать класс **InputController**.

```
public void Update()
{
    if (Input.GetKeyDown(KeyCode.F))
    {
        _isActiveFlashlight = !_isActiveFlashlight;
        if (_isActiveFlashlight)
        {
            Main.Instance.GetFlashlightController.On();
        }
        else
        {
            Main.Instance.GetFlashlightController.Off();
        }
    }
}
```

Итоги

На этом уроке мы создали основной скелет программы:

- Создали главный класс, из которого будем брать ссылки на другие классы;
- Определили, что группам объектов будут соответствовать свои базовые классы;
- Определили, что у каждого объекта будет собственный поведенческий класс или класс-контейнер;
- Создали базовый класс для всех типов объектов, который будет кэшировать данные;
- Определили, что управлять данными будут контроллеры;
- Создали базовый контроллер.

Домашнее задание

1. Расставить объекты на сцене. Для каждой группы объектов должен быть написан свой класс, с присущими для этого типа свойствами. Пополнить список общих свойств в базовом классе.
2. Добавить функционал в контроллер фонарика:
 - a. Время работы фонарика (при выключении заряд батареи восстанавливается);
 - b. Вывод на экран заряда батареи;
3. *Добавить функции для изменения свойств объектов: например, изменения слоя объекта, заморозку физического объекта по определенной оси, включение/выключение физики объекта.
4. *Добавить свой контроллер: например, для выделения объектов.

Дополнительные материалы

1. <https://github.com/neuecc/UniRx> – еще один вариант проектирования игры, основанной на событиях, без использования стандартной функции Update.
2. <https://github.com/sschmid/Entitas-CSharp> – хорошие примеры оптимизации программного кода.
3. Джозеф Хокинг, Unity в действии. Мультиплатформенная разработка на C# – очень интересная и полезная книга по основам Unity3D.
4. <https://habrahabr.ru/post/303562/> – о кэшировании данных.
5. <http://netcoder.ru/blog/csharp/240.html> – и еще статья про кэширование.
6. <https://msdn.microsoft.com/ru-ru/library/ff926074.aspx> – правила написания кода c#.
7. <https://habrahabr.ru/post/26077/> – рекомендации по написанию кода.

Используемая литература

1. Эндрю Троелсен, Филипп Джепикс: Язык программирования C# 6.0 и платформа .NET 4.6
2. Джозеф Хокинг: Unity в действии. Мультиплатформенная разработка на C#