Интерфейсы и обобщения

Интерфейсы. Стандартные интерфейсы. Обобщения. Индексаторы

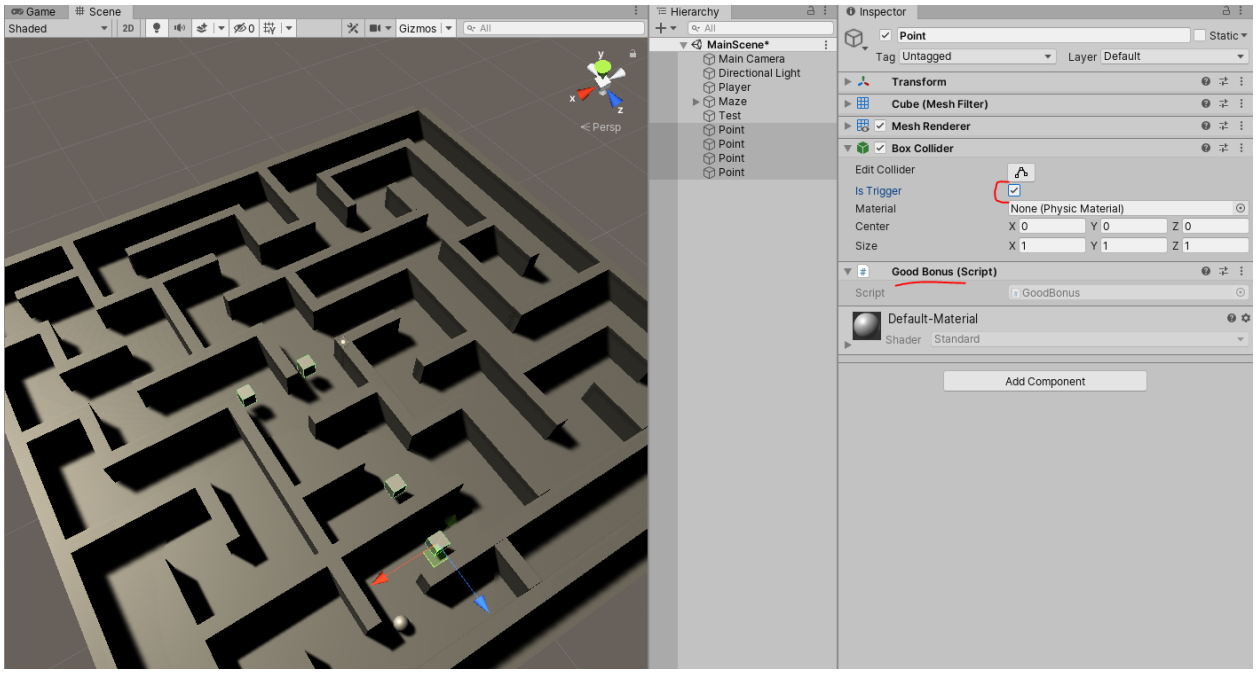
На этом уроке:

1. Вы узнаете что такое интерфейс
2. Разберем стандартные интерфейсы
3. Научимся писать свои
4. Узнаете что такое обобщения
5. Разберем индексаторы
6. Познакомитесь с фишками языка C#

Изменения в игре Roll a ball

Добавление интерактивных объектов

Класс InteractiveObject разработанный на прошлом занятии относледуем от MonoBehaviour. Создадим несколько игровых объектов и добавим на них скрипт GoodBonus. У коллайдера поставить флажок IsTrigger, для того чтобы можно их подбирать.



Интерфейс

Интерфейс похож на класс, но он содержит спецификацию, а не реализацию своих членов. Особенности интерфейсов:

- Члены интерфейса всегда неявно абстрактные. Класс, напротив, может содержать как абстрактные, так и конкретные методы с реализацией;
- Класс (или структура) может реализовать несколько интерфейсов. Класс может быть наследником только одного класса, а структура не допускает наследования вообще (кроме класса **System.ValueType**).

Интерфейсы могут определять следующие сущности:

- Методы
- Свойства
- Индексаторы
- События
- Статические поля и константы (начиная с версии C# 8.0)

Объявление интерфейса напоминает объявление класса, но не содержит реализации своих членов, поскольку все его члены неявно абстрактные. Эти члены можно реализовать в классах и структурах, реализующих интерфейс. Интерфейс может содержать только методы, свойства, события и индексаторы, которые не случайно являются членами класса (он может быть абстрактным). Для определения интерфейса используется ключевое слово **interface**. Как правило, названия интерфейсов в C# начинаются с заглавной буквы **I** (это не обязательное требование, а стиль программирования). Все члены интерфейса имеют модификаторов доступа по умолчанию public. Но с C# 8.0, мы можем явно указывать модификаторы доступа у компонентов интерфейса.

Рассмотрим пример. Предположим, что в игре объекты с которыми будет взаимодействовать пользователь могут иметь различные свойства (летать вверх-вниз, вращаться, мерцать и т.д.). Нужно заметить, что не все объекты могут обладать этими свойствами, а некоторые объекты могут обладать сразу несколькими свойствами. Для каждого свойства напишем интерфейс.

```
namespace Geekbrains
{
    public interface IFlay
    {
        void Flay();
    }
}
```

```
namespace Geekbrains
{
    public interface IRotation
    {
        void Rotation();
    }
}
```

```
namespace Geekbrains
{
    public interface IFlicker
    {
        void Flicker();
    }
}
```

И реализуем данные интерфейсы

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class GoodBonus : InteractiveObject, IFlay, IFlicker
    {
        private Material _material;
        private float _lengthFlay;
```

```
        private void Awake()
        {
            _material = GetComponent<Renderer>().material;
            _lengthFlay = Random.Range(1.0f, 5.0f);
        }

        protected override void Interaction()
        {
            // Add bonus
        }

        public void Flay()
        {
            transform.localPosition = new Vector3(transform.localPosition.x,
                Mathf.PingPong(Time.time, _lengthFlay),
                transform.localPosition.z);
        }

        public void Flicker()
        {
            _material.color = new Color(_material.color.r, _material.color.g,
_material.color.b,
                Mathf.PingPong(Time.time, 1.0f));
        }
    }
}
```

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class BadBonus : InteractiveObject, IFlay, IRotation
    {
        private float _lengthFlay;
        private float _speedRotation;

        private void Awake()
        {
            _lengthFlay = Random.Range(1.0f, 5.0f);
            _speedRotation = Random.Range(10.0f, 50.0f);
        }

        protected override void Interaction()
        {
            // Destroy player
        }

        public void Flay()
        {
            transform.localPosition = new Vector3(transform.localPosition.x,
                Mathf.PingPong(Time.time, _lengthFlay),
                transform.localPosition.z);
        }

        public void Rotation()
        {
            transform.Rotate(Vector3.up * (Time.deltaTime * _speedRotation),
Space.World);
        }
```

```
    }
}
```

Хорошим тоном будет если в приложении одна точка входа. У нас это будет класс GameController. Также для оптимизации будет неплохо иметь всего один update.

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class GameController : MonoBehaviour
    {
        private InteractiveObject[] _interactiveObjects;

        private void Awake()
        {
            _interactiveObjects = FindObjectsOfType<InteractiveObject>();
        }

        private void Update()
        {
            for (var i = 0; i < _interactiveObjects.Length; i++)
            {
                var interactiveObject = _interactiveObjects[i];

                if (interactiveObject == null)
                {
                    continue;
                }

                if (interactiveObject is IFlay flay)
                {
                    flay.Flay();
                }
                if (interactiveObject is IFlicker flicker)
                {
                    flicker.Flicker();
                }
                if (interactiveObject is IRotation rotation)
                {
                    rotation.Rotation();
                }
            }
        }
    }
}
```

Начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию.

```
using UnityEngine;

namespace Geekbrains
{
    public interface IFlay
    {
        void Flay()
        {
            Debug.Log("I can flay");
        }
    }
}
```

Интерфейсы, как и классы, могут наследоваться:

```
namespace Geekbrains
{
    public interface IAction
    {
        void Action();
    }
}
```

```
namespace Geekbrains
{
    public interface IInteractable : IAction
    {
        bool IsInteractable { get; }
    }
}
```

```
using UnityEngine;
using Random = UnityEngine.Random;

namespace Geekbrains
{
    public abstract class InteractiveObject : MonoBehaviour, IInteractable
    {
        public bool IsInteractable { get; }
        protected abstract void Interaction();

        private void Start()
        {
            Action();
        }

        public void Action()
        {
            if (TryGetComponent(out Renderer renderer))
            {
                renderer.material.color = Random.ColorHSV();
            }
        }
    }
}
```

Рассмотрим случай, когда в 2 разных интерфейсах определены функции с одинаковыми именами:

```
namespace Geekbrains
{
    public interface IInitialization
    {
        void Action();
    }
}
```

```
namespace Geekbrains
{
    public interface IInteractable : IAction, IInitialization
```

```
    {
        bool IsInteractable { get; }
    }
}
```

Класс **InteractiveObject** определяет один метод **Action()**, создавая общую реализацию для обоих примененных интерфейсов. И вне зависимости от того, будем ли мы рассматривать объект **InteractiveObject** как объект типа **IAction** или **IInitialization**, результат метода будет один и тот же.

Но нередко бывает необходимо разграничить реализуемые интерфейсы. В этом случае надо явным образом применить интерфейс:

```
using UnityEngine;
using Random = UnityEngine.Random;

namespace Geekbrains
{
    public abstract class InteractiveObject : MonoBehaviour, IInteractable
    {
        public bool IsInteractable { get; }
        protected abstract void Interaction();

        private void Start()
        {
            ((IAction)this).Action();
            ((IInitialization)this).Action();
        }

        void IAction.Action()
        {
            if (TryGetComponent(out Renderer renderer))
            {
                renderer.material.color = Random.ColorHSV();
            }
        }

        void IInitialization.Action()
        {
            if (TryGetComponent(out Renderer renderer))
            {
                renderer.material.color = Color.cyan;
            }
        }
    }
}
```

При явной реализации указывается название метода вместе с названием интерфейса. При этом мы не можем использовать модификатор **public**, то есть методы являются закрытыми. В этом случае при использовании метода **Action** в программе надо привести объект к типу соответствующего интерфейса.

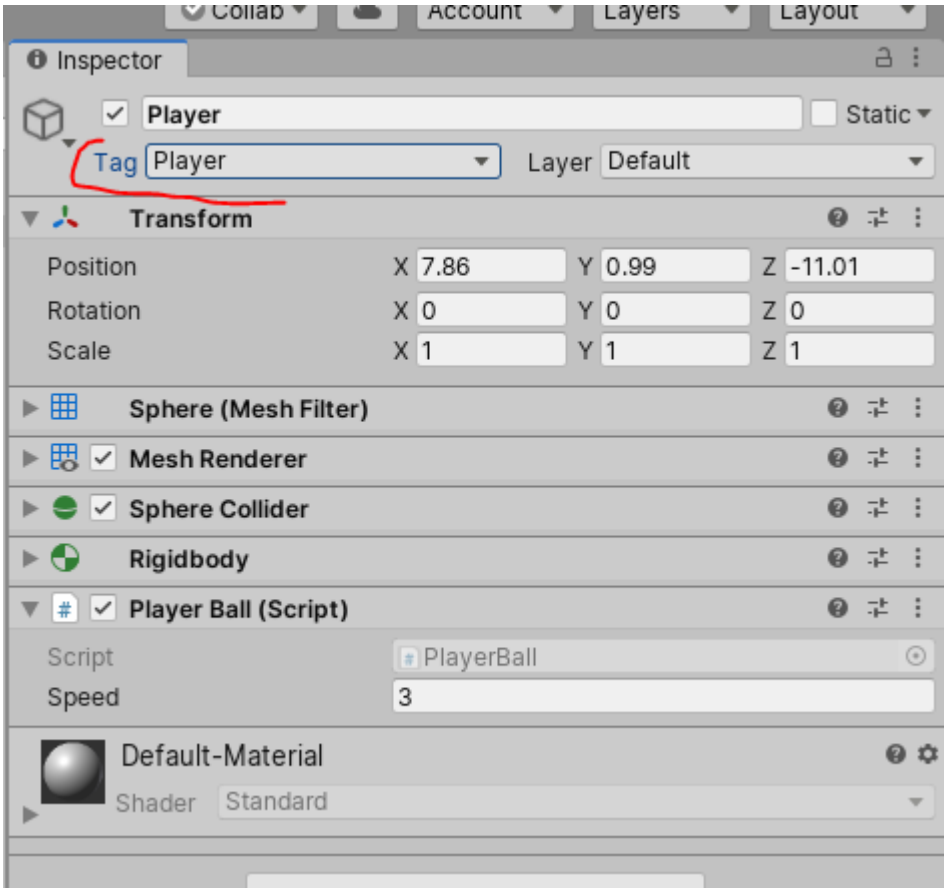
Добавление бонусов в игру

Реализуем получение бонусных баллов. Для этого сделаем объекты интерактивными добавив метод OnTriggerEnter в класс InteractiveObject. Данный метод будет срабатывать, когда объект будет интерактивный и будет соприкосновение с объектом с тэгом “Player”.

```
public abstract class InteractiveObject : MonoBehaviour, IInteractable
{
    public bool IsInteractable { get; } = true;

    private void OnTriggerEnter(Collider other)
    {
        if (!IsInteractable || !other.CompareTag("Player"))
        {
            return;
```

```
        }
        Interaction();
        Destroy(gameObject);
    }

    //.....
}
```

И не забываем у игрока выставить нужный тэг.



Напишем класс для вывода набранных очков

```
using UnityEngine.UI;
using UnityEngine;

namespace Geekbrains
{
    public sealed class DisplayBonuses
    {
        private Text _text;
        public DisplayBonuses()
        {
            _text = Object.FindObjectOfType<Text>();
        }

        public void Display(int value)
        {
            _text.text = $"Вы набрали {value}";
        }
    }
}
```

В классе GoodBonus создадим экземпляр класса DisplayBonuses и при подборе бонуса будем выводить на экран количество бонусов.

```
using UnityEngine;

namespace Geekbrains
{
    public sealed class GoodBonus : InteractiveObject, IFlay, IFlicker
    {
        private DisplayBonuses _displayBonuses;

        private void Awake()
        {
            _displayBonuses = new DisplayBonuses();
        }

        protected override void Interaction()
        {
            _displayBonuses.Display(5);
            // Add bonus
        }
    }
}
```

В таблице выше представлен неполный код класса GoodBonus

Обобщения

Обобщенное программирование (generic programming) – парадигма программирования, заключающаяся в таком описании данных и алгоритмов, которое можно применять к различным типам данных, не меняя само описание. Продемонстрируем пример необобщенной реализации функции обмена значениями двух переменных **Swap** с использованием перегруженных методов. Используя перегруженные методы, в C# мы можем создавать функции с одинаковыми именами, но с разными параметрами:

```
private void Swap(ref int A,ref  int B)
{
    int t;
    t = A;
    A = B;
    B = t;
}

private void Swap(ref  double A, ref  double
B)
{
    double t;
    t = A;
    A = B;
    B = t;
}

private void Swap(ref object a,ref object b)
{
    object t = a;
    a = b;
    b = t;
}
```

Пример реализации **Swap** при помощи обобщения:

```
private void Swap<T>(ref  T A, ref  T B) // Обобщенные методы не могут иметь out параметры
{
    T t;
    t = A;
    A = B;
    B = t;
}
```

После появления первого выпуска платформы **.NET** программисты часто использовали пространство имен **System.Collections**, чтобы получить более гибкий способ управления данными в приложениях. Но с версии .NET 2.0 язык программирования C# был расширен поддержкой обобщения (generic). Вместе с ним библиотеки базовых классов пополнились совершенно новым пространством имен, связанным с коллекциями – **System.Collections.Generic**.

Термин «обобщение», по сути, означает параметризированный тип. Особая роль параметризированных типов состоит в том, что они позволяют создавать классы, структуры, интерфейсы, методы и делегаты, в которых обрабатываемые данные указываются в виде параметра. С помощью обобщений можно создать единый класс, который автоматически становится пригодным для обработки разнотипных данных. Класс, структура, интерфейс, метод или делегат, оперирующий параметризованным типом данных, называется обобщенным (обобщенный класс или метод).

В C# всегда была возможность создавать обобщенный код, оперируя ссылками типа **object**. А поскольку класс **object** является базовым для всех остальных классов, по ссылке типа **object** можно обращаться к объекту любого типа. Таким образом, до появления обобщений для оперирования разнотипными объектами в программах служил обобщенный код, в котором для этой цели использовались ссылки типа **object**.

В таком коде трудно было соблюсти типовую безопасность, поскольку для преобразования типа **object** в конкретный тип данных требовалось приведение типов. Это потенциальный источник ошибок: приведение типов могло быть неумышленно выполнено неверно. Обобщения решили проблему типовой безопасности. Они упростили и процесс в целом, поскольку исключили необходимость выполнять приведение типов для преобразования объекта или другого типа обрабатываемых данных. Таким образом, обобщения расширяют возможности повторного использования кода и позволяют делать это надежно и просто.

Использование обобщений

Разберем обобщение на примере класса, который будет использоваться в дальнейшем для сохранения прогресса игры

```
namespace Geekbrains
{
    public sealed class SavedData
    {
        public int CountBonuses;
        public object IdPlayer;
    }
}
```

```
using System;
using UnityEngine;

namespace Geekbrains
{
```

```
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            SavedData savedData = new SavedData();
            savedData.IdPlayer = new Guid();
            savedData.IdPlayer = "name_235";
        }
    }
}
```

Такое решение – не из оптимальных. Дело в том, что в данном случае мы сталкиваемся с такими явлениями, как упаковка (boxing) и распаковка (unboxing). Как мы уже упоминали, упаковка (boxing) 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) преобразует объект типа **object** к значимому типу. Упаковка и распаковка ведут к снижению производительности, так как системе надо выполнять преобразования. Кроме того, существует проблема безопасности типов.

```
namespace Geekbrains
{
    public sealed class SavedData<T>
    {
        public int CountBonuses;
        public T IdPlayer;
    }
}
```

```
using System;
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            var savedData = new SavedData<Guid>
            {
                IdPlayer = new Guid()
            };

            var savedDataExample = new SavedData<string>
            {
                IdPlayer =  "name_235"
            };
        }
    }
}
```

Используя букву **T** в описании класса **SavedData<T>**, мы указываем, что данный тип будет использоваться этим классом. В классе мы создаем поле данного типа. Причем сейчас нам неизвестно, что это будет за тип – и он может быть любым. А параметр **T** в угловых скобках – это «параметр универсального типа», так как вместо него можно подставить любой тип.

Иногда возникает необходимость присвоить переменным универсальных параметров начальное значение, в том числе и **null**. Но напрямую мы его присвоить не можем. В этом случае надо использовать оператор **default(T)**. Он присваивает ссылочным типам в качестве значения **null**, а типам значений – значение **0**:

```
namespace Geekbrains
{
    public sealed class SavedData<T>
    {
        public int CountBonuses;
        public T IdPlayer = default;
    }
}
```

При типизации обобщенного класса определенным типом будет создаваться свой набор статических членов:

```
namespace Geekbrains
{
    public sealed class SavedData<T>
    {
        public int CountBonuses;
        public static T IdPlayer = default;
    }
}
```

Для **SavedData<string>** и для **SavedData<Guid>** будет создана своя переменная **IdPlayer**.

Обобщения могут использовать несколько универсальных параметров, которые могут представлять различные типы, одновременно:

```
using System;
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            var savedData = new SavedData<Guid, Vector3>
            {
                IdPlayer = new Guid(),
                Position = Vector3.one
            };
        }
    }
}
```

Ограничения обобщений

Чтобы конкретизировать тип, применяется контекстно-зависимое ключевое слово **where**:

```
namespace Geekbrains
{
    public sealed class SavedData<T> where T : struct
    {
        public int CountBonuses;
        public T IdPlayer = default;
    }
}
```

```
where T : struct - аргумент типа должен быть структурного типа, кроме Nullable.
where T : class - аргумент типа должен иметь ссылочный тип; это также
распространяется на тип любого класса, интерфейса, делегата или массива.
where T : <base class name> - аргумент типа должен являться или быть производным от
указанного базового класса
where T : U - аргумент типа, поставляемый для T, должен являться или быть
производным от аргумента, поставляемого для U. Это называется неприкрытым
ограничением типа
where T : new () - ограничение указывает, что аргумент любого типа в объявлении
универсального класса должен иметь открытый конструктор без параметров.
Использовать ограничение new можно только в том случае, если тип не является
абстрактным
```

Наследование обобщенных типов

Обобщения также поддерживают различные варианты наследования.

Первый вариант – когда производный класс типизирован тем же типом, что и базовый:

```
namespace Geekbrains
{
    public sealed class SavedData<T> : BaseExample<T>
    {
        public SavedData(T id) : base(id)
        {
            IdPlayer = id;
        }
    }

    public class BaseExample<T>
    {
        public T IdPlayer = default;

        public BaseExample(T id)
        {
            IdPlayer = id;
        }
    }
}
```

```
using System;
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            BaseExample<string> data= new BaseExample<string>("name_235");
            BaseExample<int> data1 = new SavedData<int>(3775);
            SavedData<Guid> data2 = new SavedData<Guid>(Guid.NewGuid());
        }
    }
}
```

Второй вариант – создание обычного необобщенного класса. В этом случае при наследовании у базового класса надо явным образом определить используемый тип:

```
using System;

namespace Geekbrains
{
    public sealed class SavedData : BaseExample<Guid>
    {
        public SavedData(Guid id) : base(id)
        {
            IdPlayer = id;
        }
    }

    public class BaseExample<T>
    {
        public T IdPlayer = default;

        public BaseExample(T id)
        {
            IdPlayer = id;
        }
    }
}
```

```
using System;
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            BaseExample<string> data= new BaseExample<string>("name_235");
            BaseExample<Guid> data1 = new SavedData(Guid.NewGuid());
            SavedData data2 = new SavedData(Guid.NewGuid());
        }
    }
}
```

Третий вариант – когда типизация класса-наследника отличается от универсального параметра в базовом классе. В этом случае для базового класса также надо указать используемый тип:

```
using System;

namespace Geekbrains
{
    public sealed class SavedData<T>  : BaseExample<Guid>
    {
        public T Position;
        public SavedData(Guid id) : base(id)
        {
            IdPlayer = id;
        }
    }

    public class BaseExample<T>
    {
        public T IdPlayer = default;
```

```
        public BaseExample(T id)
        {
            IdPlayer = id;
        }
    }
}
```

```
using System;
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            BaseExample<string> data= new BaseExample<string>("name_235");
            BaseExample<Guid> data1 = new SavedData<Vector3>(Guid.NewGuid());
            SavedData<Vector3> data2 = new SavedData<Vector3>(Guid.NewGuid());
        }
    }
}
```

В производных классах можно сочетать использование универсального параметра из базового класса с применением своих параметров:

```
namespace Geekbrains
{
    /// <summary>
    /// Аргумент типа, предоставляемый в качестве T, должен совпадать с аргументом,
предоставляемым в качестве U, или быть производным от него.
    /// </summary>
    /// <typeparam name="T"></typeparam>
    /// <typeparam name="R"></typeparam>
    /// <typeparam name="U"></typeparam>
    public sealed class SavedData<T, R, U> : BaseExample<R>
        where T : U
    {
        public T Position;
        public SavedData(R id) : base(id)
        {
            IdPlayer = id;
        }
    }

    public class BaseExample<T>
    {
        public T IdPlayer = default;

        public BaseExample(T id)
        {
            IdPlayer = id;
        }
    }
}
```

```
using System;
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            BaseExample<string> data = new BaseExample<string>("name_235");
            BaseExample<Guid> data1 = new SavedData<int, Guid,
object>(Guid.NewGuid());
            SavedData<int, Guid, object> data2 = new SavedData<int, Guid,
object>(Guid.NewGuid());
        }
    }
}
```

Стандартные интерфейсы

Интерфейс IEnumerator и IEnumerable

Рассмотрим упрощенную версию интерфейса **IEnumerator**, определенную в классе **System. Collections**:

Оператор **foreach** применяется только к классам, в которых реализован интерфейс **IENumerator**. Этот пример демонстрирует важность интерфейсов. Они задают только то, что должен выполнять класс, но не говорят, как это делается. Оператор **foreach** может проходить по элементам класса, потому что интерфейс **IEnumerator** обязывает реализовать в классе переход к следующему элементу и получение значения текущего элемента. Все классы, с которыми вы ранее использовали оператор **foreach**, являются наследниками интерфейса **IEnumerator**.

```
using System.Collections;
using UnityEngine;

namespace Geekbrains
{
    public sealed class ListInteractableObject : IEnumerator
    {
        private InteractiveObject[] _interactiveObjects;
        private int _index = -1;

        public ListInteractableObject()
        {
            _interactiveObjects = Object.FindObjectsOfType<InteractiveObject>();
        }

        public bool MoveNext()
        {
            if (_index == _interactiveObjects.Length - 1)
            {
                Reset();
                return false;
            }

            _index++;
            return true;
```

```
        }

        public void Reset() => _index = -1;

        public object Current => _interactiveObjects[_index];
    }
}
```

```
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            var interactableObject = new ListInteractableObject();

            while (interactableObject.MoveNext())
            {
                print(interactableObject.Current);
            }

        }
    }
}
```

Для того чтобы можно было пробежаться циклом foreach, нужно реализовать интерфейс IEnumerable

```
using System.Collections;
using UnityEngine;

namespace Geekbrains
{
    public sealed class ListInteractableObject : IEnumerator, IEnumerable
    {
        private InteractiveObject[] _interactiveObjects;
        private int _index = -1;
        private InteractiveObject _current;

        public ListInteractableObject()
        {
            _interactiveObjects = Object.FindObjectsOfType<InteractiveObject>();
        }

        public bool MoveNext()
        {
            if (_index == _interactiveObjects.Length - 1)
            {
                Reset();
                return false;
            }

            _index++;
            return true;
        }
```

```
        public void Reset() => _index = -1;

        public object Current => _interactiveObjects[_index];

        public IEnumerator GetEnumerator()
        {
            return this;
        }

        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }
    }
}
```

```
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            var interactableObject = new ListInteractableObject();

            foreach (var o in interactableObject)
            {
                print(o);
            }
        }
    }
}
```

Интерфейс IComparable. Сортировка по одному критерию

Во многих классах приходится реализовывать интерфейс **IComparable**, поскольку он позволяет сравнивать один объект с другим, используя различные методы, определенные в среде **.NET Framework**.

Интерфейс **IComparable** реализуется чрезвычайно просто, потому что он состоит всего лишь из одного метода:

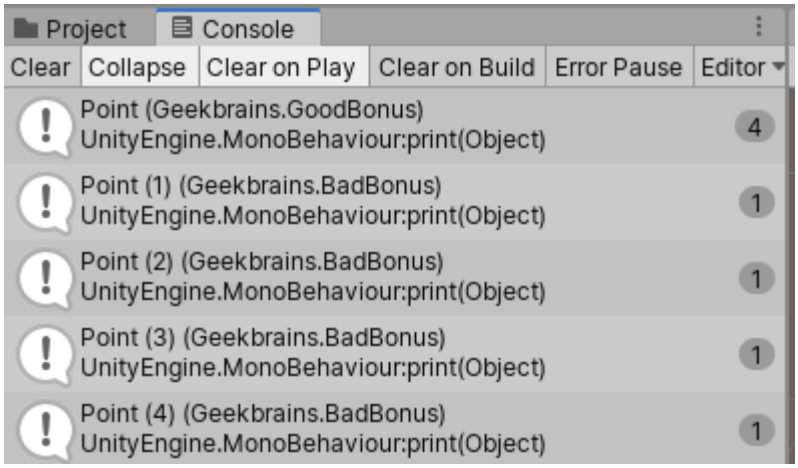
```
int CompareTo(object obj)
```

В этом методе значение вызывающего объекта сравнивается со значением объекта, определяемого параметром **obj**. Если значение вызывающего объекта больше, чем у объекта **obj**, то возвращается положительное значение; если оба значения равны – нулевое значение; если значение вызывающего объекта меньше, чем у объекта **obj** – отрицательное значение.

```
using System;
using UnityEngine;
using Random = UnityEngine.Random;

namespace Geekbrains
{
    public abstract class InteractiveObject : MonoBehaviour, IInteractable,
IComparable<InteractiveObject>
```

```
    {
        //....
        public int CompareTo(InteractiveObject other)
        {
            return name.CompareTo(other.name);
        }
    }
}
```

```
1 usage
public sealed class ListInteractableObject : IEnumerator, IEnumerable
{
    private InteractiveObject[] _interactiveObjects;
    private int _index = -1;
    private InteractiveObject _current;

    1 usage
    public ListInteractableObject()
    {
        _interactiveObjects = Object.FindObjectsOfType<InteractiveObject>();
        Array.Sort(_interactiveObjects);
    }
```



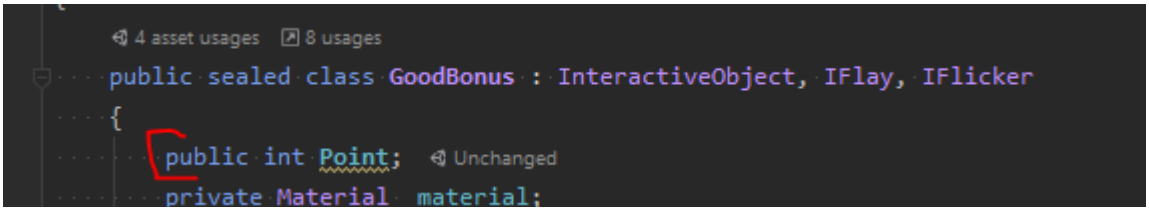
Интерфейс IComparer. Сортировка по разным критериям

Интерфейс **IComparer** определен в пространстве имен **System.Collections**. Он содержит один метод **CompareTo**, возвращающий результат сравнения двух объектов, переданных ему в качестве параметров:

```
int Compare(object x, object y)
```

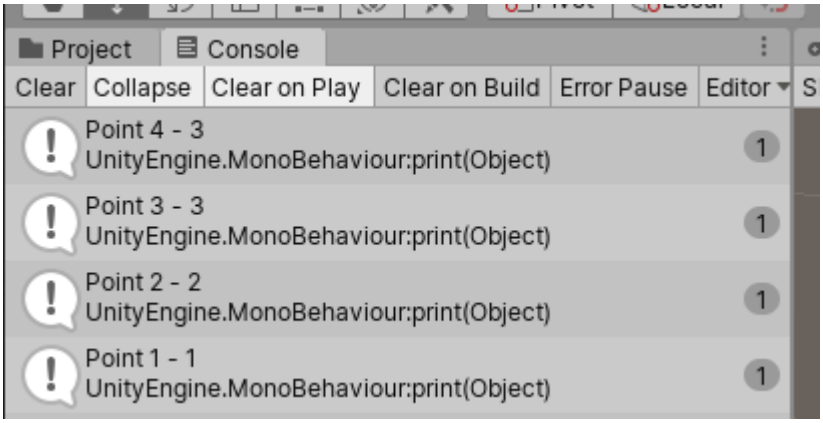
Принцип применения этого интерфейса состоит в том, что для каждого критерия сортировки объектов описывается небольшой вспомогательный класс, реализующий этот интерфейс. Объект этого класса передается в стандартный метод сортировки массива в качестве второго аргумента (существует несколько перегруженных версий этого метода).

Добавим в класс GoodBonus поле Point, которое будет хранить очки, которые будут начисляться пользователю. Создадим класс для сортировки объектов по количеству очков.



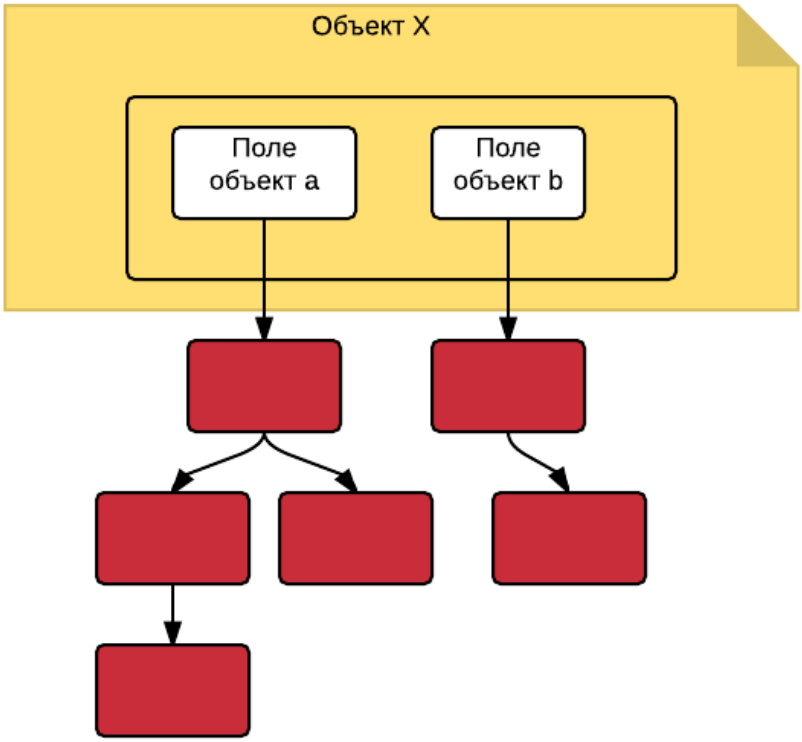

```cs
using System.Collections.Generic;

namespace Geekbrains
{
    public sealed class GoodBonusComparer : IComparer<GoodBonus>
    {
        public int Compare(GoodBonus x, GoodBonus y)
        {
            if (x.Point < y.Point)
            {
                return 1;
            }

            if (x.Point > y.Point)
            {
                return -1;
            }

            return 0;
        }
    }
}
```

```cs
using System.Linq;
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            var goodBonusComparer = new GoodBonusComparer();
            var objects = FindObjectsOfType<GoodBonus>().ToList();
            objects.Sort(goodBonusComparer);
            foreach (var goodBonus in objects)
            {
                print($"{goodBonus.name} - {goodBonus.Point}");
            }
        }
    }
}
```



Клонирование объектов (интерфейс ICloneable)

Клонирование – это создание копии объекта (клона). При присваивании одного объекта ссылочного типа другому, копируется ссылка, а не сам объект. Если необходимо скопировать в другую область памяти поля объекта, можно воспользоваться методом **MemberwiseClone**, который любой объект наследует от класса **object**. При этом объекты (ссылки), на которые указывают поля объекта, не копируются. Это называется поверхностным клонированием.



Для создания полностью независимых объектов необходимо глубокое клонирование, когда в памяти создается дубликат всего дерева объектов (тех, на которые ссылаются поля объекта, поля полей и так далее). Алгоритм глубокого клонирования сложен, поскольку требует рекурсивного обхода всех ссылок объекта и отслеживания циклических зависимостей.

Пример реализации интерфейса **ICloneable**:

```
using System;
using UnityEngine;
using Random = UnityEngine.Random;

namespace Geekbrains
{
    public sealed class BadBonus : InteractiveObject, IFlay, IRotation, ICloneable
    {
        //....
        public object Clone()
        {
            var result = Instantiate(gameObject, transform.position,
transform.rotation, transform.parent);
            return result;
        }
    }
}
```

```
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            FindObjectOfType<BadBonus>().Clone();
        }
    }
}
```

Рассмотрим еще один пример. Допустим, у нас есть класс для хранения информации о пользователе:

```
using System;

namespace Geekbrains
{
    public sealed class ExampleDeepCopy
    {
        public sealed class Person : ICloneable
        {
            public string Name { get; set; }
            public int Age { get; set; }

            public object Clone()
            {
                return new Person
                {
                    Name = Name,
                    Age = Age
                };
            }
        }
    }
}
```

Для сокращения кода копирования мы можем использовать специальный метод **MemberwiseClone()**, который возвращает копию объекта:

```
public object Clone() => MemberwiseClone();
```

Этот метод реализует поверхностное (неглубокое) копирование, но его может быть недостаточно. Например, пусть класс **Person** содержит ссылку на объект **Company**. Поверхностное копирование работает только для свойств, представляющих примитивные типы, но не для сложных объектов. И в этом случае надо применять глубокое копирование:

```
using System;

namespace Geekbrains
{
    public sealed class ExampleDeepCopy
    {
        public sealed class Person : ICloneable
        {
            public string Name { get; set; }
            public int Age { get; set; }
            public Company Work { get; set; }

            public object Clone()
            {
                Company company = new Company { Name = Work.Name };
                return new Person
                {
                    Name = Name,
                    Age = Age,
                    Work = company
                };
            }
        }

        public sealed class Company
        {
            public string Name { get; set; }
        }
    }
}
```

Dispose

Метод финализации может применяться для освобождения *неуправляемых* ресурсов при активизации процесса сборки мусора. Однако многие неуправляемые объекты являются «ценными элементами» (например, низкоуровневые соединения с базой данных или файловые дескрипторы). Зачастую выгоднее освобождать их как можно раньше, еще до наступления момента сборки мусора. Поэтому вместо переопределения **Finalize()** в качестве альтернативного варианта также можно реализовать в классе интерфейс **IDisposable**, который имеет единственный метод – **Dispose()**.

```
using System;
using UnityEngine;

namespace Geekbrains
{
    public sealed class GameController : MonoBehaviour, IDisposable
    {
```

```
        private InteractiveObject[] _interactiveObjects;

        private void Awake()
        {
            _interactiveObjects = FindObjectsOfType<InteractiveObject>();
        }

        private void Update()
        {
            for (var i = 0; i < _interactiveObjects.Length; i++)
            {
                var interactiveObject = _interactiveObjects[i];

                if (interactiveObject == null)
                {
                    continue;
                }

                if (interactiveObject is IFlay flay)
                {
                    flay.Flay();
                }
                if (interactiveObject is IFlicker flicker)
                {
                    flicker.Flicker();
                }
                if (interactiveObject is IRotation rotation)
                {
                    rotation.Rotation();
                }
            }
        }

        public void Dispose()
        {
            foreach (var o in _interactiveObjects)
            {
                Destroy(o.gameObject);
            }
        }
    }
}
```

Когда действительно реализуется поддержка интерфейса **IDisposable**, предполагается, что после завершения работы с объектом метод **Dispose()** должен вручную вызываться пользователем этого объекта прежде, чем объектной ссылке будет позволено покинуть область действия. Благодаря этому объект может выполнять любую необходимую очистку неуправляемых ресурсов без попадания в очередь финализации и без ожидания того, когда сборщик мусора запустит содержащуюся в классе логику финализации. Объекты **IDisposable**, как правило, работают в связке с конструкцией **using**.

```
using System.IO;
using UnityEngine;

namespace Geekbrains
{
    public sealed class Test : MonoBehaviour
    {
        private void Start()
```

```
        {
            using (StreamReader reader = new StreamReader("example path"))
            {
                // Работаем с объектом
            }   // Неявно вызвана функция Dispose()
            // Объект выгружен из памяти
        }
    }
}
```

Есть полная реализация паттерна **IDisposable.** Но как правило бывает достаточно простой реализации метода **Dispose.**

```
using System;

namespace Geekbrains
{
    public abstract class DisposeExample : IDisposable
    {
        private bool _disposed;

        // реализация интерфейса IDisposable.
        public void Dispose()
        {
            Dispose(true);
            // подавляем финализацию
            GC.SuppressFinalize(this);
        }

        protected virtual void Dispose(bool disposing)
        {
            if (!_disposed)
            {
                if (disposing)
                {
                    // Освобождаем управляемые ресурсы
                }
                // освобождаем неуправляемые объекты
                _disposed = true;
            }
        }

        // Деструктор
        ~DisposeExample()
        {
            Dispose(false);
        }
    }

    public sealed class Derived : DisposeExample
    {
        private bool _isDisposed = false;

        protected override void Dispose(bool disposing)
        {
            if (!_isDisposed)
            {
                if (disposing)
                {
                    // Освобождение управляемых ресурсов
```

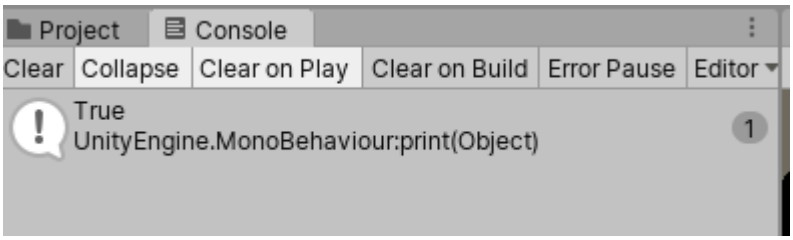
```
                }
                _isDisposed = true;
            }
            // Обращение к методу Dispose базового класса
            base.Dispose(disposing);
        }
    }
}
```

IEquatable

Интерфейс IEquatable необходим для сравнения объектов по различным характеристикам. В основном применяется в структурах данных.

```
using System;
using UnityEngine;
using Random = UnityEngine.Random;

namespace Geekbrains
{
    public sealed class GoodBonus : InteractiveObject, IFlay, IFlicker,
IEquatable<GoodBonus>
    {
        public int Point;
        //.....

        public bool Equals(GoodBonus other)
        {
            return Point == other.Point;
        }
    }
}
```

```
using System.Linq;
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            var objects = FindObjectsOfType<GoodBonus>().ToList();
            var bonus = GameObject.Find("Point 4").GetComponent<GoodBonus>();
            objects.Remove(bonus);
            print($"{objects.Contains(bonus)}");
        }
    }
}
```



На сцене находятся 2 бонуса, которые дают 3 победных очка, Point 3 и Point 4. Один объект мы удаляем.

IEqualityComparer

Этот интерфейс позволяет реализовать настраиваемое сравнение на равенство для коллекций. То есть можно создать собственное определение равенства для типа T

```
using System.Collections.Generic;

namespace Geekbrains
{
    public sealed class GoodBonusesEqualityComparer : IEqualityComparer<GoodBonus>
    {
        public bool Equals(GoodBonus x, GoodBonus y) => x.Point == y.Point;

        public int GetHashCode(GoodBonus obj) => obj.Point.GetHashCode();
    }
}
```

```
using System.Linq;
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            var objects = FindObjectsOfType<GoodBonus>().Distinct(new
GoodBonusesEqualityComparer());
            foreach (var o in objects)
            {
                print($"{o.name}");
            }
        }
    }
}
```

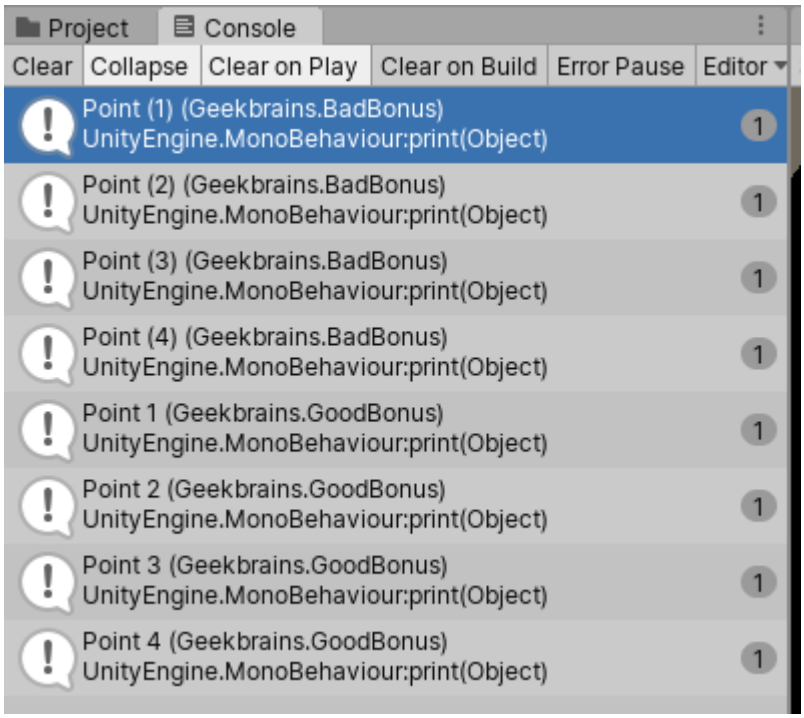
Индексаторы

Индексаторы позволяют индексировать экземпляры класса или структуры точно так же, как и массивы. Индексированное значение можно задавать или получать без явного указания типа или экземпляра элемента. Индексаторы действуют как свойства, за исключением того, что их акцессоры принимают параметры.

```
возвращаемый_тип this [Тип параметр1, ...]
{
    get { ... }
    set { ... }
}
```

```
using System;
using System.Collections;
using Object = UnityEngine.Object;

namespace Geekbrains
{
    public sealed class ListInteractableObject : IEnumerator, IEnumerable
    {
        private InteractiveObject[] _interactiveObjects;
        private int _index = -1;
        private InteractiveObject _current;

        public ListInteractableObject()
        {
            _interactiveObjects = Object.FindObjectsOfType<InteractiveObject>();
            Array.Sort(_interactiveObjects);
        }

        public InteractiveObject this [int index]
        {
            get => _interactiveObjects[index];
            set => _interactiveObjects[index] = value;
        }

        public int Count => _interactiveObjects.Length;

        //...
    }
}
```

```
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            var objects = new ListInteractableObject();

            for (int i = 0; i < objects.Count; i++)
            {
                print($"{objects[i]}");
            }
        }
    }
}
```



Статический импорт

С версии 6.0 в язык C# была добавлена возможность импорта функциональности классов:

```
using UnityEngine;
using static UnityEngine.Time;
using static UnityEngine.Mathf;
using static UnityEngine.Random;

namespace Geekbrains
{
    public sealed class BadBonus : InteractiveObject, IFlay, IRotation
    {
        private float _lengthFlay;
        private float _speedRotation;

        private void Awake()
        {
            _lengthFlay = Range(1.0f, 5.0f);
            _speedRotation = Range(10.0f, 50.0f);
        }

        protected override void Interaction()
        {
            // Destroy player
        }

        public void Flay()
        {
            transform.localPosition = new Vector3(transform.localPosition.x,
                PingPong(time, _lengthFlay),
                transform.localPosition.z);
        }
```

```
        public void Rotation()
        {
            transform.Rotate(Vector3.up * (deltaTime * _speedRotation),
Space.World);
        }
    }
}
```

.

Практическое задание

1. Расписать в текстовом документе зачем нужны интерфейсы и их отличие от классов
2. Расписать в текстовом документе зачем нужны обобщение и привести несколько примеров их использования
3. Добавить в игру положительные бонусы, которые необходимо обязательно собрать для победы
4. Добавить в игру положительные бонусы, которые дают различные улучшения (временный прирост скорости, неуязвимость и т.д.)
5. Добавить в игру различные ловушки (моментальная смерть, замедление и т.д.)
6. Реализовать статический импорт класса Debug, где это необходимо
7. * Реализовать интерфейс ICloneable для клонирование game object
8. * Реализовать интерфейс IDisposable для игрока, ловушек и бонусов

Дополнительные материалы

1. https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/interfaces/
2. https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/generics/generic-classes

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.