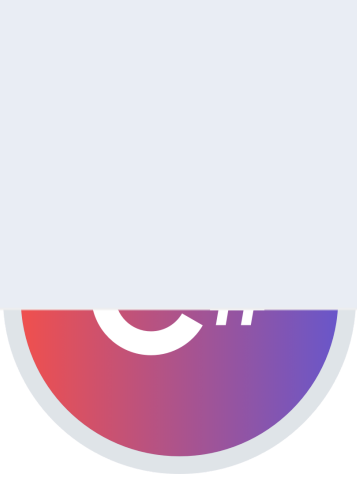


Урок 3

Делегаты и события. Исключения

Делегаты и события. Исключения.

Делегаты

- Использование делегатов
- Передача делегатов в методы
- Групповая адресация
- Удаление целей из списка вызовов делегата

Обобщенные делегаты

- Обобщенные делегаты Action<>, Func<> и Predicate<>
- Паттерн «наблюдатель»

События

Встроенные события Unity

Исключительная ситуация

Обработка исключений

Генерация собственных исключений

Советы по работе с исключениями

Практическое задание

Дополнительные материалы

Используемая литература

На этом уроке:

1. Вы узнаете что такое делегаты
2. Разберем и научимся использовать события
3. Научимся работать с исключительной ситуацией

Делегаты

Делегат – это вид класса, предназначенный для хранения ссылок на методы. Как и любой другой класс, его можно передать в качестве параметра, а затем вызвать содержащийся в нем метод по ссылке. Делегаты используются для поддержки событий, а также как самостоятельная конструкция языка.

Пример описания делегата:

```
public delegate int ExampleDelegate(int i,int j);
```

Здесь описан тип делегата, который может хранить ссылки на методы, возвращающие **int** и принимающие два параметра типа **int**.

Объявление делегата можно размещать непосредственно в пространстве имен или внутри класса. Чтобы вызвать метод, на который указывает делегат, надо использовать его метод **Invoke**.

Использование делегатов

Чтобы воспользоваться делегатом, необходимо создать его экземпляр и задать имена методов, на которые он будет ссылаться. При вызове экземпляра делегата вызываются все заданные в нем методы.

Делегаты применяются в основном для следующих целей:

- Чтобы определять вызываемый метод не при компиляции, а динамически во время выполнения программы;
- Обеспечивать связь между объектами по типу «источник-наблюдатель»;
- Создавать универсальные методы, в которые можно передавать другие методы;
- Поддерживать механизм обратных вызовов.

Передача делегатов в методы

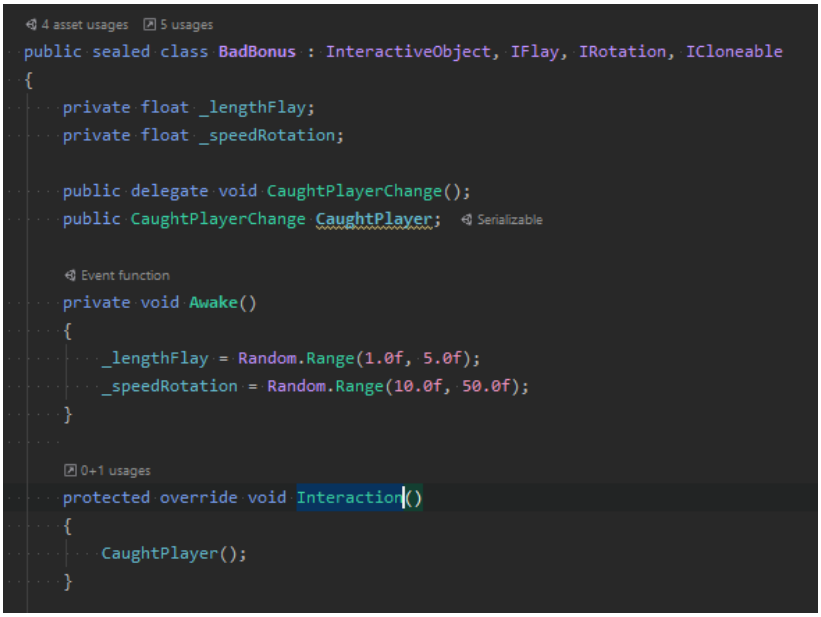
Поскольку делегат является классом, его можно передавать в методы в качестве параметра. Таким образом обеспечивается функциональная параметризация: в метод можно передавать не только данные, но и функции их обработки.

Простейший пример универсального метода – метод вывода таблицы значений функции, в который передается диапазон значений аргумента, шаг его изменения и вид вычисляемой функции:

Групповая адресация

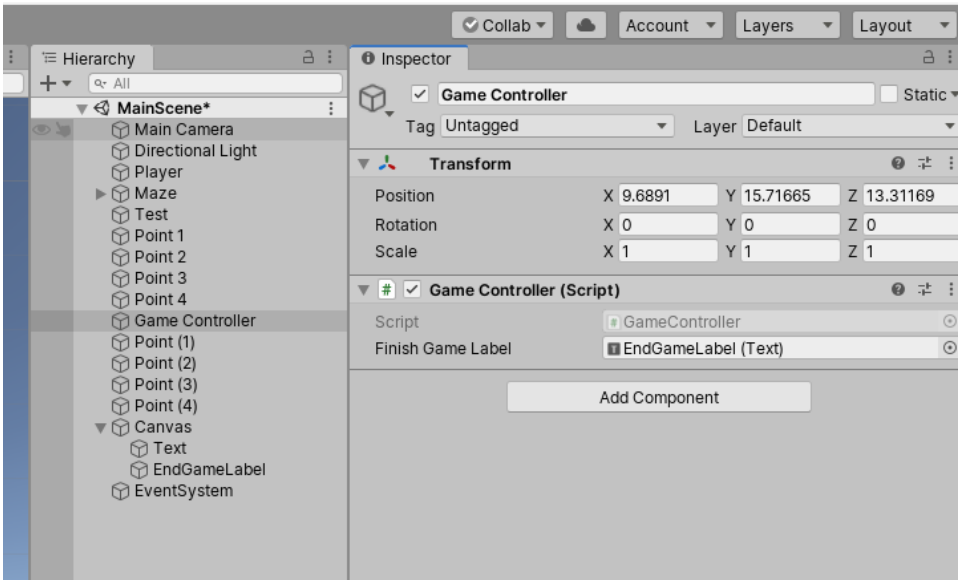
Делегаты **.NET** обладают встроенной возможностью группового вызова. Другими словами, объект делегата может поддерживать целый список методов для вызова. Для добавления нескольких методов к объекту делегата используется перегруженная операция **+=**, а не прямое присваивание.

В классе **BadBonus** добавим делегат и вызовем его в методе **Interaction**



Добавим класс **DisplayEndGame**, который будет выводить сообщение о конце игры

```
using UnityEngine.UI;

namespace Geekbrains
{
    public sealed class DisplayEndGame
    {
        private Text _finishGameLabel;

        public DisplayEndGame(Text finishGameLabel)
        {
            _finishGameLabel = finishGameLabel;
            _finishGameLabel.text = String.Empty;
        }

        public void GameOver()
        {
            _finishGameLabel.text = "Вы проиграли";
        }
    }
}
```



В классе **GameController** добавим ссылки на делегат класса **BadBonus**

```
using System;
using UnityEngine;
using UnityEngine.UI;

namespace Geekbrains
{
    public sealed class GameController : MonoBehaviour, IDisposable
    {
        public Text _finishGameLabel;
        private ListInteractableObject _interactiveObject;
        private DisplayEndGame _displayEndGame;

        private void Awake()
        {
            _interactiveObject = new ListInteractableObject();
            _displayEndGame = new DisplayEndGame(_finishGameLabel);
            foreach (var o in _interactiveObject)
            {
                if (o is BadBonus badBonus)
                {
                    badBonus.CaughtPlayer += CaughtPlayer;
                    badBonus.CaughtPlayer += _displayEndGame.GameOver;
                }
            }
        }

        private void CaughtPlayer()
        {
            Time.timeScale = 0.0f;
        }

        private void Update()
        {
            for (var i = 0; i < _interactiveObject.Length; i++)
            {
                var interactiveObject = _interactiveObject[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 _interactiveObject)
            {
                if (o is InteractiveObject interactiveObject)
                {
                    Destroy(interactiveObject.gameObject);
                }
            }
        }
    }
}
```

Удаление целей из списка вызовов делегата

Для удаления метода из списка делегатов можно воспользоваться перегруженной операцией **-=**.

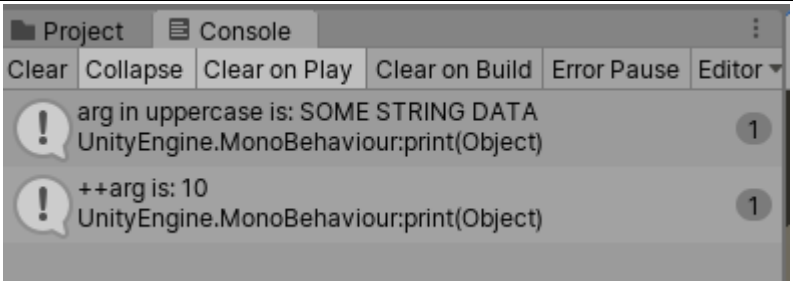
Удаление метода:

```
public void Dispose()
{
    foreach (var o in _interactiveObject)
    {
        if (o is InteractiveObject interactiveObject)
        {
            Destroy(interactiveObject.gameObject);
            if (o is BadBonus badBonus)
            {
                badBonus.CaughtPlayer -= CaughtPlayer;
                badBonus.CaughtPlayer -= _displayEndGame.GameOver;
            }
        }
    }
}
```

Обобщенные делегаты

Язык C# позволяет определять обобщенные типы делегатов. Например, необходимо определить делегат, который может вызывать любой метод, возвращающий **void** и принимающий единственный параметр. Если передаваемый аргумент может изменяться – это моделируется через параметр типа.

```
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
// Этот обобщенный делегат может вызывать любой метод, который возвращает void и
принимает
//  единственный параметр типа
        private delegate void MyGenericDelegate<T>(T arg);
        private void Start()
        {
// Зарегистрировать цели
            MyGenericDelegate<string> strTarget = new
MyGenericDelegate<string>(StringTarget);
            strTarget("Some string data");
// А можно просто указать метод
            MyGenericDelegate<int> intTarget = IntTarget;
            intTarget(9);
        }

        void StringTarget(string arg)
        {
            print($"arg in uppercase is: {arg.ToUpper()}");
        }

        void IntTarget(int arg)
        {
            print($"++arg is: {++arg}");
        }
    }
}
```



Обобщенные делегаты Action<>, Func<> и Predicate<>

Когда необходимо использовать делегаты для включения обратных вызовов в приложениях, ранее мы выполняли следующие шаги:

- определение специального делегата, соответствующего формату метода, на который он указывает;

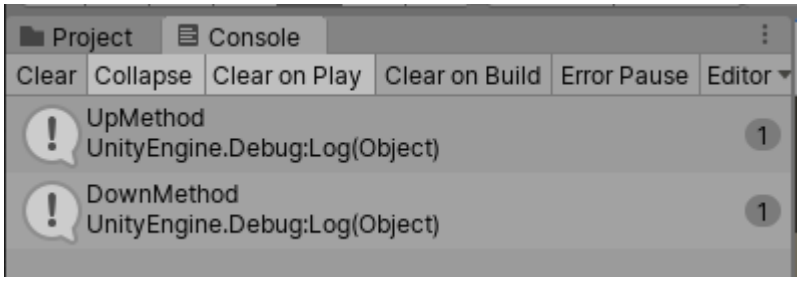
- создание экземпляра специального делегата с передачей имени метода в качестве аргумента конструктора;
- косвенное обращение к методу через вызов **Invoke ()** на объекте делегата.

При таком подходе в конечном итоге, как правило, получается несколько специальных делегатов, которые никогда не могут применяться за пределами текущей задачи (например, **MyGenericDelegate<T>**, **CarEngineHandler** и т.д.). Может случаться так, что в проекте требуется специальный, уникально именованный делегат, но в других ситуациях точное имя делегата несущественно. Во многих случаях необходим просто «некоторый делегат», принимающий набор аргументов и, возможно, возвращающий значение, отличное от **void**. В таких ситуациях можно воспользоваться встроенными в платформу делегатами **Action<>** и **Func<>**.
Обобщенный делегат **Action<>** определен в пространствах имен **System** внутри сборок **mscorlib.dll** и **System.Core.dll**. Его можно применять для указания на метод, который принимает вплоть до 16 аргументов (этого должно быть достаточно) и возвращает **void**. Поскольку **Action<>** является обобщенным делегатом, понадобится также указывать типы каждого параметра.

Пример замены оператора switch:

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

namespace Geekbrains
{
    public enum UserAction : byte
    {
        None = 0,
        Up   = 1,
        Down = 2
    }

    public sealed class ActionDelegateExample
    {
        private readonly Dictionary<UserAction, Action> _actions;

        public ActionDelegateExample()
        {
            _actions = new Dictionary<UserAction, Action>
            {
                [UserAction.Up] = UpMethod,
                [UserAction.Down] = DownMethod
            };
        }

        public Action this[UserAction index] => _actions[index];

        private void UpMethod()
        {
            Debug.Log(nameof(UpMethod));
        }

        private void DownMethod()
        {
            Debug.Log(nameof(DownMethod));
        }
    }
}
```

```
using UnityEngine;

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

            example[UserAction.Up]();
            example[UserAction.Down]();
        }
    }
}
```



В примере использовалось некая структура данных Dictionary, о которой мы поговорим в следующей методичке.

Применение **Action<>** не заставляет беспокоиться об определении специального делегата. Но он может указывать только на методы, которые имеют возвращаемое значение **void**. Если нужно указывать на метод с другим возвращаемым значением (и нет желания заниматься написанием собственного делегата), можно прибегнуть к делегату **Func<>**.

Обобщенный делегат **Func<>** может указывать на методы, которые (подобно **Action<>**) принимают вплоть до 16 параметров и имеют специальное возвращаемое значение.

Делегаты **Action<>** и **Func<>** могут устранить шаг по ручному определению специального делегата. Учитывая это, следует ли ими пользоваться всегда? Это зависит от ситуации. Во многих случаях **Action<>** и **Func<>** будут предпочтительным вариантом. Тем не менее, если нужен делегат со специфическим именем, которое, как вы чувствуете, помогает лучше отразить предметную область, то построение специального делегата сведется к одиночному оператору кода.

Делегат **Predicate<T>**, как правило, используется для сравнения, сопоставления некоторого объекта **T** определенному условию. В качестве выходного результата возвращается значение **true**, если условие соблюдено, и **false**, если не соблюдено:

```
using System;
using UnityEngine;

namespace Geekbrains
{
    public sealed class PredicateAndFuncDelegatesExample : MonoBehaviour
    {
        public Predicate<Collision> Predicate;
```

```
        public Func<float, float> Func;
        private float _armor = 3.0f;
        private float _hp = 100.0f;
        private void OnCollisionEnter(Collision other)
        {
            if (Predicate(other))
            {
                _hp = Func(_armor);
            }
            Debug.Log(_hp);
        }
    }
}
```

```
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            var example = FindObjectOfType<PredicateAndFuncDelegatesExample>();
            example.Predicate = collision =>
collision.gameObject.CompareTag("Player");
            const float damage = 50;
            example.Func = f => f - damage;
        }
    }
}
```

Паттерн «наблюдатель»

Для обеспечения связи между объектами во время выполнения программы применяется следующая стратегия. Объект-источник при изменении своего состояния, которое значимо для других объектов, посылает им уведомления. Эти объекты называются наблюдателями. Получив уведомления, наблюдатель опрашивает источник, чтобы синхронизировать с ним свое состояние.

Наблюдатель (observer) определяет между объектами зависимость типа «один ко многим», так что при изменении состоянии одного объекта все зависящие от него объекты получают извещение и автоматически обновляются.

Рассмотрим пример:

```
using System;
namespace Delegates_Observer
{
using UnityEngine;

namespace Geekbrains
{
    public sealed class DelegatesObserver
    {
        public delegate void MyDelegate(object o);
        public sealed class Source
        {
            private MyDelegate _functions;
```

```
            public void Add(MyDelegate f)
            {
                _functions += f;
            }

            public void Remove(MyDelegate f)
            {
                _functions -= f;
            }

            public void Run()
            {
                Debug.Log("RUNS!");
                if (_functions != null) _functions(this);
            }
        }

        public sealed class Observer1 // Наблюдатель 1
        {
            public void Do(object o)
            {
                Debug.Log($"Первый. Принял, что объект {o} побежал");
            }
        }

        public sealed class Observer2 // Наблюдатель 2
        {
            public void Do(object o)
            {
                Debug.Log($"Второй. Принял, что объект {o} побежал");
            }
        }
    }
}
```

```
using UnityEngine;

 namespace Geekbrains
 {
     public class Test : MonoBehaviour
     {
         private void Start()
         {
             DelegatesObserver.Source s = new DelegatesObserver.Source();
             DelegatesObserver.Observer1 o1 = new DelegatesObserver.Observer1();
             DelegatesObserver.Observer2 o2 = new DelegatesObserver.Observer2();
             DelegatesObserver.MyDelegate d1 = new
DelegatesObserver.MyDelegate(o1.Do);
             s.Add(d1);
             s.Add(o2.Do);
             s.Run();
             s.Remove(o1.Do);
             s.Run();
         }
     }
 }
```

События

Рассмотрим подробнее события среды **.Net Framework**.

Событие – это элемент класса, позволяющий ему посылать другим объектам уведомления об изменении своего состояния. При этом для объектов, которые являются наблюдателями события, активизируются методы-обработчики этого события.

События построены на основе делегатов: с их помощью вызываются методы-обработчики событий. Поэтому создание события в классе состоит из следующих частей:

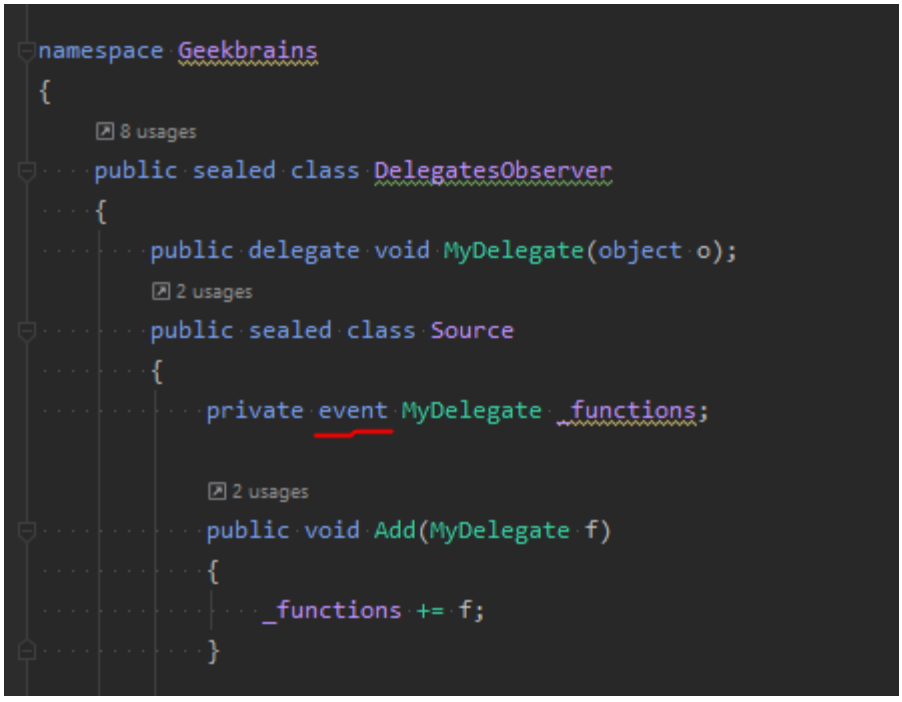
- описание делегата, задающего сигнатуру обработчиков событий;
- описание события;
- описание методов, инициирующих событие.

Пример описания делегата и соответствующего ему события:

```
public delegate void Delegat();
class A
{
    public event Delegate Event;
}
```

Событие – это удобная абстракция для программиста. На самом деле, событие состоит из закрытого статического класса, в котором создается экземпляр делегата, и двух методов, предназначенных для добавления и удаления обработчика из списка этого делегата.

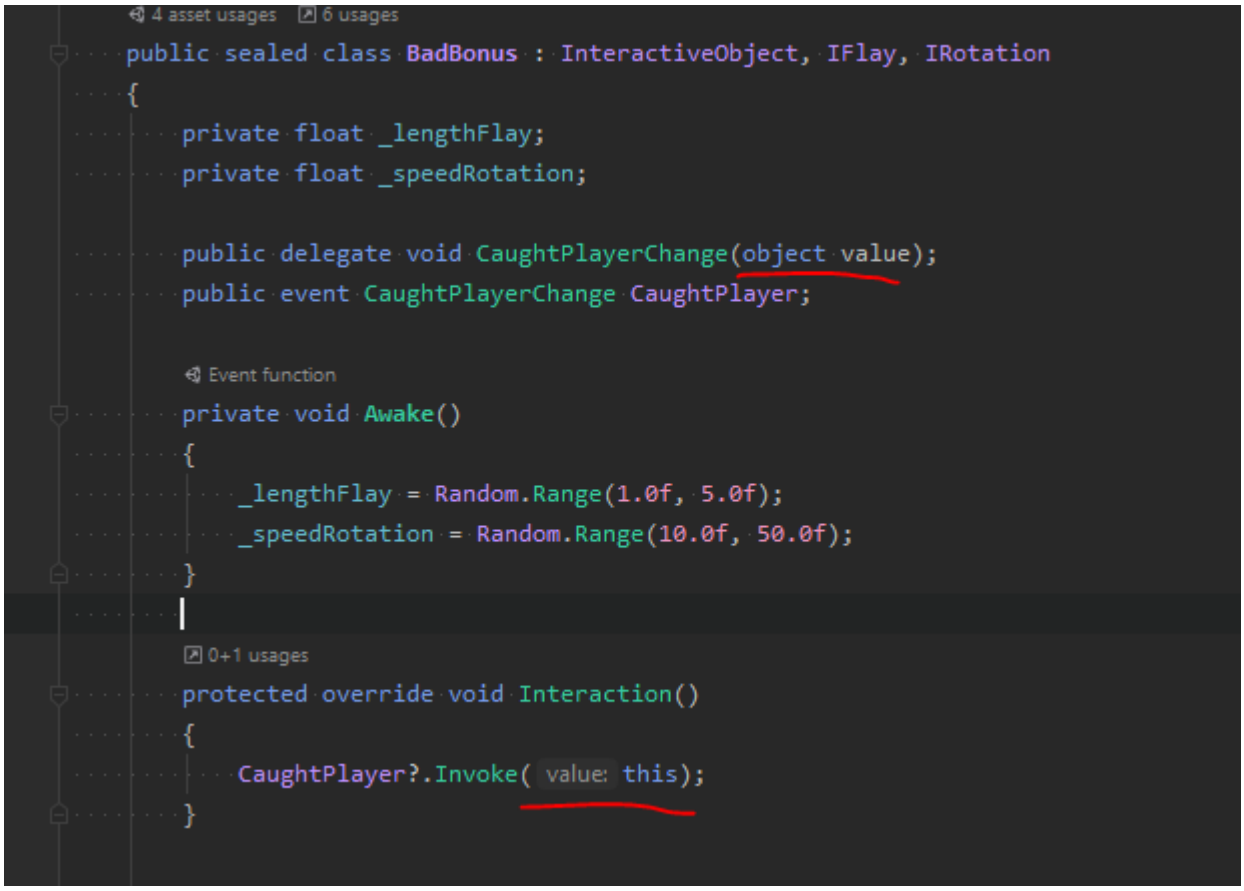
Давайте разберем еще один пример



Пример создания событий

```
using UnityEngine;

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

        public delegate void CaughtPlayerChange();
        public event CaughtPlayerChange CaughtPlayer;

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

        protected override void Interaction()
        {
            CaughtPlayer?.Invoke();
        }

        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);
        }
    }
}
```

Данный пример демонстрирует передачу данных между классами. Но, как правило, нам нужно знать, от кого пришло данное событие.



```
4 asset usages  6 usages
public sealed class BadBonus : InteractiveObject, IFlay, IRotation
{
    private float _lengthFlay;
    private float _speedRotation;

    public delegate void CaughtPlayerChange(object value);
    public event CaughtPlayerChange CaughtPlayer;

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

    0+1 usages
    protected override void Interaction()
    {
        CaughtPlayer?.Invoke( value: this);
    }
```

```
using System;
using UnityEngine.UI;

namespace Geekbrains
{
    public sealed class DisplayEndGame
    {
        private Text _finishGameLabel;

        public DisplayEndGame(Text finishGameLabel)
        {
            _finishGameLabel = finishGameLabel;
            _finishGameLabel.text = String.Empty;

        }

        public void GameOver(object o)
        {
            _finishGameLabel.text = "Вы проиграли";
        }
    }
}
```

```
using System;
using UnityEngine;
using UnityEngine.UI;

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

```
{
    public Text _finishGameLabel;
    private ListInteractableObject _interactiveObject;
    private DisplayEndGame _displayEndGame;

    private void Awake()
    {
        _interactiveObject = new ListInteractableObject();
        _displayEndGame = new DisplayEndGame(_finishGameLabel);
        foreach (var o in _interactiveObject)
        {
            if (o is BadBonus badBonus)
            {
                badBonus.CaughtPlayer += CaughtPlayer;
                badBonus.CaughtPlayer += _displayEndGame.GameOver;
            }
        }
    }

    private void CaughtPlayer(object value)
    {
        Time.timeScale = 0.0f;
    }

    private void Update()
    {
        for (var i = 0; i < _interactiveObject.Length; i++)
        {
            var interactiveObject = _interactiveObject[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 _interactiveObject)
        {
            if (o is InteractiveObject interactiveObject)
            {
                if (o is BadBonus badBonus)
                {
                    badBonus.CaughtPlayer -= CaughtPlayer;
                    badBonus.CaughtPlayer -= _displayEndGame.GameOver;
```

```
                    }
                    Destroy(interactiveObject.gameObject);
                }
            }
        }
    }
}
```

Полная реализация события:

```
public delegate void CaughtPlayerChange(object value);
private event CaughtPlayerChange _caughtPlayer;
public event CaughtPlayerChange CaughtPlayer
{
    add { _caughtPlayer += value; }
    remove { _caughtPlayer -= value; }
}

Event function
private void Awake(){...}

0+1 usages
protected override void Interaction()
{
    _caughtPlayer.Invoke( value: this);
}
```

Делегат **EventHandler**.

Добавим чтобы каждый интерактивный предмет знал свой сцвет.

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

namespace Geekbrains
{
    public abstract class InteractiveObject : MonoBehaviour, IInteractable
    {
        protected Color _color;
        public bool IsInteractable { get; } = true;

        private void OnTriggerEnter(Collider other)
        {
            if (!IsInteractable || !other.CompareTag("Player"))
            {
                return;
            }
            Interaction();
            Destroy(gameObject);
        }

        protected abstract void Interaction();

        private void Start()
        {
```

```
            Action();
        }

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

Передадим цвет в событие

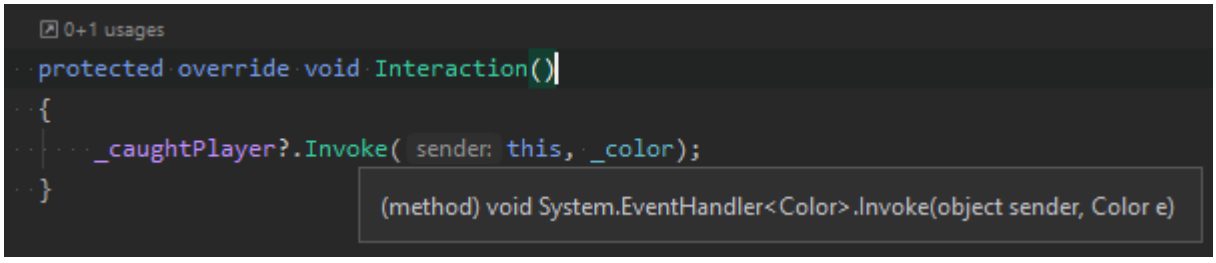
```
using System;
using UnityEngine;
using Random = UnityEngine.Random;

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

        private event EventHandler<Color> _caughtPlayer;
        public event EventHandler<Color> CaughtPlayer
        {
            add { _caughtPlayer += value; }
            remove { _caughtPlayer -= value; }
        }

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

        protected override void Interaction()
        {
            _caughtPlayer?.Invoke(this, _color);
        }

        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);
        }
    }
}
```

```
using System;
using UnityEngine;
using UnityEngine.UI;

namespace Geekbrains
{
    public sealed class DisplayEndGame
    {
        private Text _finishGameLabel;

        public DisplayEndGame(Text finishGameLabel)
        {
            _finishGameLabel = finishGameLabel;
            _finishGameLabel.text = String.Empty;

        }

        public void GameOver(object o, Color color)
        {
            _finishGameLabel.text = $"Вы проиграли. Вас убил {((GameObject)o).name}
{color} цвета";
        }
    }
}
```

```
using System;
using UnityEngine;
using UnityEngine.UI;

namespace Geekbrains
{
    public sealed class GameController : MonoBehaviour, IDisposable
    {
        public Text _finishGameLabel;
        private ListInteractableObject _interactiveObject;
        private DisplayEndGame _displayEndGame;

        private void Awake()
        {
            _interactiveObject = new ListInteractableObject();
            _displayEndGame = new DisplayEndGame(_finishGameLabel);
            foreach (var o in _interactiveObject)
            {
                if (o is BadBonus badBonus)
                {
                    badBonus.CaughtPlayer += CaughtPlayer;
                    badBonus.CaughtPlayer += _displayEndGame.GameOver;
                }
            }
        }

        private void CaughtPlayer(object value, Color color)
        {
            Time.timeScale = 0.0f;
        }

        private void Update()
```

```
        {
            for (var i = 0; i < _interactiveObject.Length; i++)
            {
                var interactiveObject = _interactiveObject[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 _interactiveObject)
            {
                if (o is InteractiveObject interactiveObject)
                {
                    if (o is BadBonus badBonus)
                    {
                        badBonus.CaughtPlayer -= CaughtPlayer;
                        badBonus.CaughtPlayer -= _displayEndGame.GameOver;
                    }
                    Destroy(interactiveObject.gameObject);
                }
            }
        }
    }
}
```

Мы избавились от делегата, но потеряли читабельность.



Неочевидно, какие параметры нужно передавать. Поэтому мы введем дополнительный класс:

```
using System;
using UnityEngine;

namespace Geekbrains
```

```
{
    public sealed class CaughtPlayerEventArgs : EventArgs
    {
        public Color Color { get; }
        // Можем дописать сколько угодно свойств
        public CaughtPlayerEventArgs(Color Color)
        {
            Color = Color;
        }

    }
}
```

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

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

        private event EventHandler<CaughtPlayerEventArgs> _caughtPlayer;
        public event EventHandler<CaughtPlayerEventArgs> CaughtPlayer
        {
            add { _caughtPlayer += value; }
            remove { _caughtPlayer -= value; }
        }

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

        protected override void Interaction()
        {
            _caughtPlayer?.Invoke(this, new CaughtPlayerEventArgs(_color));
        }

        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);
        }
    }
}
```

```
using System;
using UnityEngine;
using UnityEngine.UI;

namespace Geekbrains
{
    public sealed class DisplayEndGame
    {
        private Text _finishGameLabel;

        public DisplayEndGame(Text finishGameLabel)
        {
            _finishGameLabel = finishGameLabel;
            _finishGameLabel.text = String.Empty;

        }

        public void GameOver(object o, CaughtPlayerEventArgs args)
        {
            _finishGameLabel.text = $"Вы проиграли. Вас убил {((GameObject)o).name}
{args.Color} цвета";
        }
    }
}
```

```
using System;
using UnityEngine;
using UnityEngine.UI;

namespace Geekbrains
{
    public sealed class GameController : MonoBehaviour, IDisposable
    {
        public Text _finishGameLabel;
        private ListInteractableObject _interactiveObject;
        private DisplayEndGame _displayEndGame;

        private void Awake()
        {
            _interactiveObject = new ListInteractableObject();
            _displayEndGame = new DisplayEndGame(_finishGameLabel);
            foreach (var o in _interactiveObject)
            {
                if (o is BadBonus badBonus)
                {
                    badBonus.CaughtPlayer += CaughtPlayer;
                    badBonus.CaughtPlayer += _displayEndGame.GameOver;
                }
            }
        }

        private void CaughtPlayer(object value, CaughtPlayerEventArgs args)
        {
            Time.timeScale = 0.0f;
        }

        private void Update()
        {
            for (var i = 0; i < _interactiveObject.Length; i++)
```

```
            {
                var interactiveObject = _interactiveObject[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 _interactiveObject)
            {
                if (o is InteractiveObject interactiveObject)
                {
                    if (o is BadBonus badBonus)
                    {
                        badBonus.CaughtPlayer -= CaughtPlayer;
                        badBonus.CaughtPlayer -= _displayEndGame.GameOver;
                    }
                    Destroy(interactiveObject.gameObject);
                }
            }
        }
    }
}
```

Анонимные методы

С делегатами тесно связано понятие анонимных методов. Анонимные методы представляют сокращенную запись методов. Иногда они нужны для обработки одного события и больше нигде не используются. Анонимные методы позволяют встроить код там, где он вызывается. Практически, это тот же самый метод, только встроенный в код. Встраивание происходит с помощью ключевого слова **delegate**, после которого идет список параметров и далее сам код анонимного метода. В отличие от блока методов или условных и циклических конструкций, блок анонимных методов должен заканчиваться точкой с запятой после закрывающей фигурной скобки. Если для анонимного метода не требуется параметров, он используется без скобок.

```
Event function
private void Awake()
{
    _interactiveObject = new ListInteractableObject();
    _displayEndGame = new DisplayEndGame(_finishGameLabel);
    foreach (var o :object in _interactiveObject)
    {
        if (o is BadBonus badBonus)
        {
            badBonus.CaughtPlayer += CaughtPlayer;
            badBonus.CaughtPlayer += _displayEndGame.GameOver;
            badBonus.CaughtPlayer += delegate (object sender, CaughtPlayerEventArgs args)
                { Debug.Log( message: $"Вы проиграли. Вас убил {((GameObject) o).name} {args.Color} цвета"); };
        }
    }
}
```

Лямбды

Лямбда-выражения представляют упрощенную запись анонимных методов. Позволяют создать емкие лаконичные методы, которые могут возвращать значение и которые можно передать в качестве параметров в другие методы.

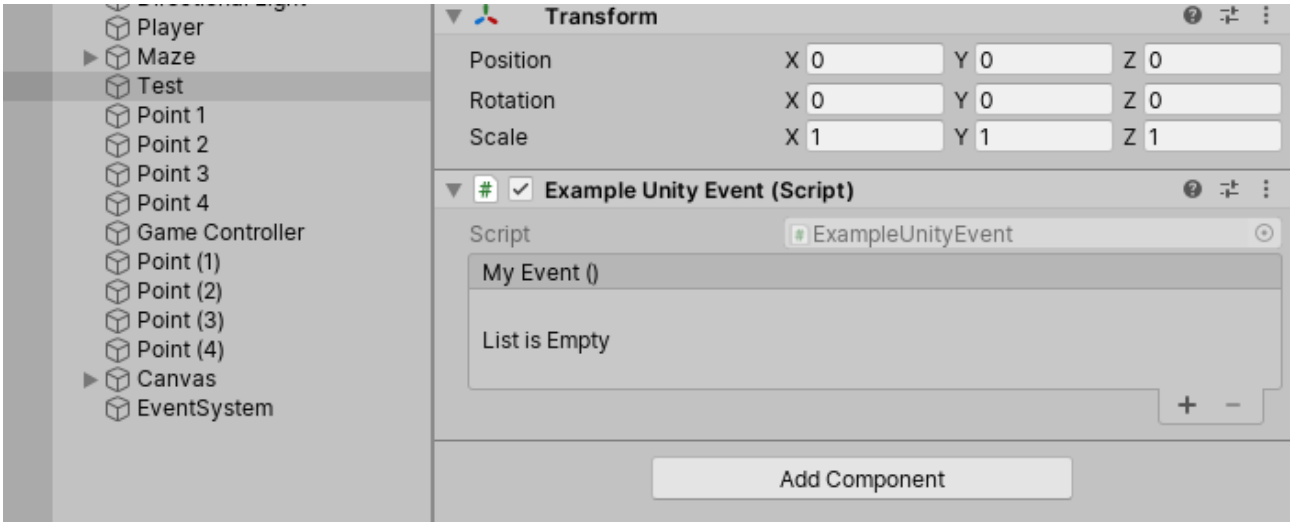
Ламбда-выражения имеют следующий синтаксис: слева от лямбда-оператора **=>** определяется список параметров, а справа – блок выражений, использующий эти параметры: **(список_параметров) => выражение**. Как и делегаты, лямбда-выражения можно передавать в качестве параметров методу:

```
Event function
private void Awake()
{
    _interactiveObject = new ListInteractableObject();
    _displayEndGame = new DisplayEndGame(_finishGameLabel);
    foreach (var o :object in _interactiveObject)
    {
        if (o is BadBonus badBonus)
        {
            badBonus.CaughtPlayer += CaughtPlayer;
            badBonus.CaughtPlayer += _displayEndGame.GameOver;
            badBonus.CaughtPlayer += (sender :object , args) =>
            {
                Debug.Log( message: $"Вы проиграли. Вас убил {((GameObject) o).name} {args.Color} цвета");
            };
        }
    }
}
```

Встроенные события Unity

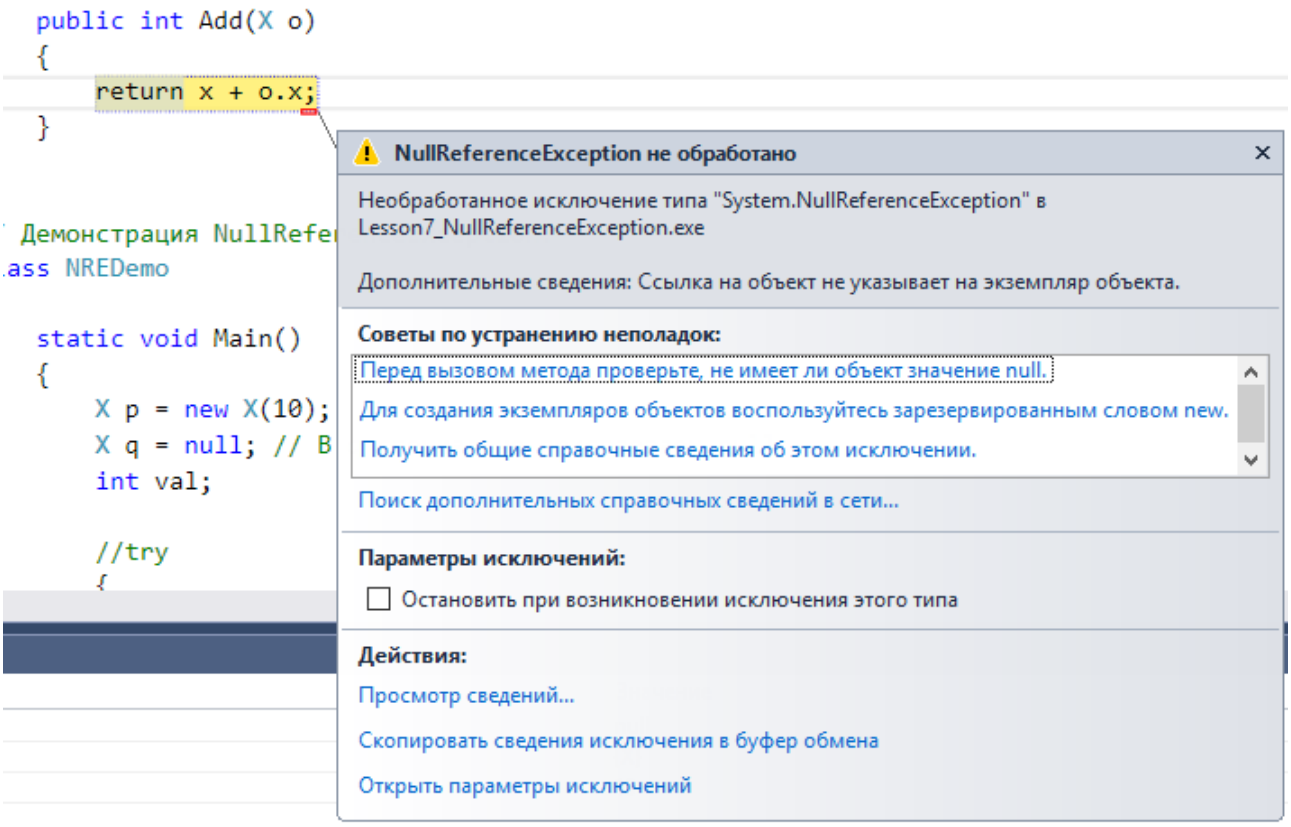
```
using UnityEngine;
using UnityEngine.Events;

namespace Geekbrains
{
```

```
    public sealed class ExampleUnityEvent : MonoBehaviour
    {
        public UnityEvent MyEvent;

        private void OnEnable()
        {
            if (MyEvent == null)
                MyEvent = new UnityEvent();

            MyEvent.AddListener(Ping);
        }

        private void OnDisable()
        {
            if (MyEvent == null)
                MyEvent = new UnityEvent();

            MyEvent.RemoveListener(Ping);
        }

        private void Update()
        {
            if (Input.anyKeyDown && MyEvent != null)
            {
                MyEvent.Invoke();
            }
        }

        private void Ping()
        {
            Debug.Log("Ping");
        }
    }
}
```



Исключительная ситуация

Скорее всего, вы уже сталкивались с исключительными ситуациями при написании программ. В C# так называется ситуация, не предусмотренная программистом. Вот пример возникновения довольно частой исключительной ситуации – ссылка на неинициализированный объект :



Среда **Visual Studio** и язык программирования C# предоставляет богатый набор возможностей для обработки исключительных ситуаций.

Обработка исключений

Язык C#, как и многие другие объектно-ориентированные языки, реагирует на ошибки и ненормальные ситуации с помощью механизма обработки исключений. Исключение – это объект, генерирующий информацию о «необычном программном происшествии». При этом важно различать ошибку в программе, ошибочную ситуацию и исключительную ситуацию.

Ошибка в программе допускается программистом при разработке. Например, вместо операции сравнения (==) используется операция присваивания (=). Программист должен исправить подобные ошибки до передачи кода программы заказчику. Механизм обработки исключений – это не защита от ошибок в программе.

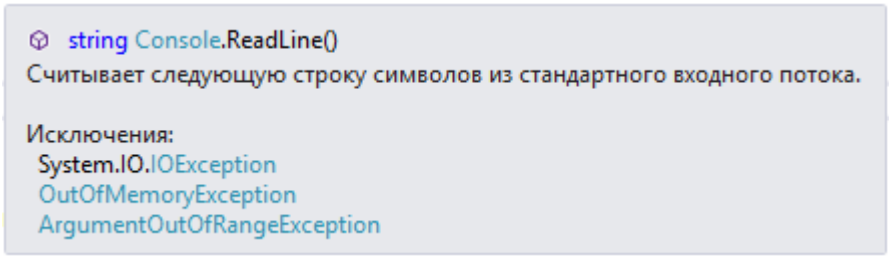
Ошибочная ситуация вызвана действиями пользователя. Например, вместо числа введена строка. Такая ошибка способна вызывать исключение. Программист должен предвидеть ошибочные ситуации и предотвращать их с помощью операторов, проверяющих допустимость поступающих данных.

Даже если программист исправил все свои баги в программе и предвидел все ошибочные ситуации, он все равно может столкнуться с непредсказуемыми и неотвратимыми проблемами – исключительными ситуациями. Например, с нехваткой доступной памяти или попыткой открыть несуществующий файл. Исключительные ситуации разработчик предвидеть не может, но может отреагировать на них так, что они не приведут к краху программы.

Для обработки ошибочных исключительных ситуаций в C# используется специальная подсистема обработки исключений. Ее преимущество – в автоматизации создания большей части кода по обработке исключений. Раньше этот код приходилось вводить в программу вручную. Обработчик

исключений также способен распознавать и выдавать информацию о таких стандартных исключениях, как деление на ноль или попадание вне диапазона определения индекса.

Visual Studio позволяет легко определить, какие исключения может выдать тот или иной метод. Для этого достаточно навести на метод мышью – и **IntelliSense** выведет описание метода со списком исключений, которые он может сгенерировать.



Рассмотрим пример перехвата исключения:

```
using System;
using System.IO;
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            StreamWriter sw = null;
            try
            {
                var path = Path.Combine(@"C:\", "temp", "text.txt");
                sw = new StreamWriter(path);
                int a;
                do
                {
                    a = Convert.ToInt32(1);
                    sw.WriteLine(a);
                } while (a != 0);
            }
            catch (FormatException)
            {
                Debug.Log("Ошибка ввода данных");
            }
            catch (IOException)
            {
                Debug.Log("Ошибка ввода/вывода");
            }
            catch (Exception exc)
            {
                Debug.Log("Неизвестная ошибка");
                Debug.Log("Информация об ошибке" + exc.Message);
            }
            finally
            {
// Использование блока finally гарантирует, что набор операторов будет выполняться
всегда, независимо от того, возникло исключение любого типа или нет)
                sw?.Close();
            }
        }
    }
}
```

Синтаксис оператора **try**:

```
try   // Контролируемый блок
{
    //...
}
catch    // Один или несколько блоков обработки исключений
{
    //...
}
finally   // Блок завершения
{
   // ...
}
```

Программные инструкции, которые нужно проконтролировать на предмет исключений, помещаются в блок **try**. Если исключение возникает в этом блоке, оно дает о себе знать выбросом определенного рода информации. Она может быть перехвачена и обработана с помощью блока **catch**. Любой код, который должен быть обязательно выполнен при выходе из блока **try**, помещается в блок **finally**.

При необходимости мы можем разграничить обработку различных типов исключений, включив дополнительные блоки:

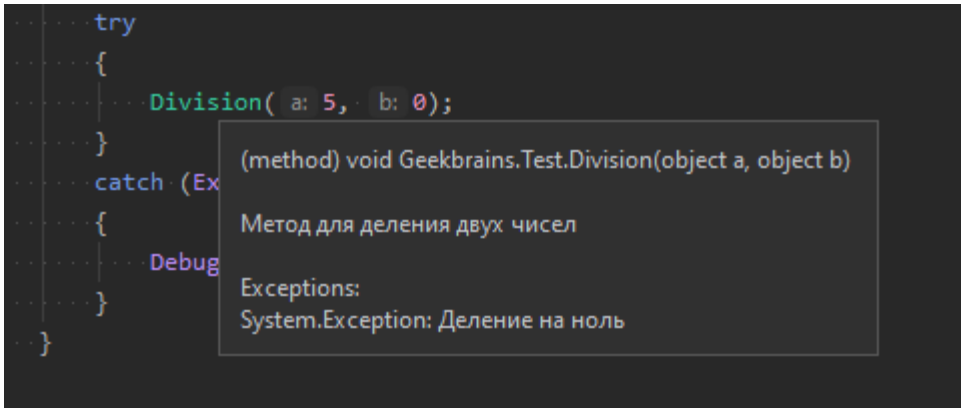
```
try
{
}
catch (FileNotFoundException e)
{
    // Обработка исключения, возникшего при отсутствии файла
}
catch (IOException e)
{
    // Обработка исключений ввода-вывода
}
catch (Exception)
{
    // Остальные исключения
}
```

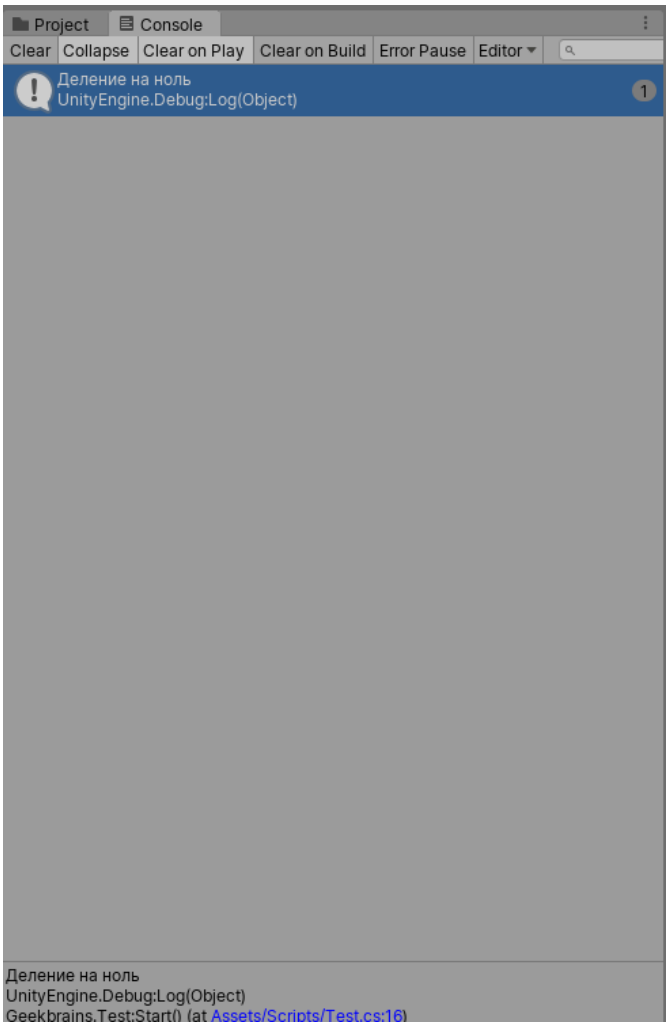
Если возникает исключение определенного типа, оно переходит к соответствующему блоку **catch**. При этом более частные исключения следует помещать в начале, и только потом – более общие классы исключений. Например, сначала обрабатывается исключение **IOException**, и затем **Exception** (так как **IOException** наследуется от класса **Exception**).

Чтобы сообщить о выполнении исключительных ситуаций в программе, можно использовать оператор **throw**. Так мы сможем пробросить исключение дальше, до ближайшего **try catch**:

```
using System;
using UnityEngine;

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

```
            try
            {
                Division(5, 0);
            }
            catch (Exception e)
            {
                Debug.Log(e.Message);
            }
        }

        /// <summary>
        /// Метод для деления двух чисел
        /// </summary>
        /// <param name="a"></param>
        /// <param name="b"></param>
        /// <exception cref="Exception">Деление на ноль</exception>
        private void Division(object a, object b)
        {
            if (Convert.ToInt32(b) == 0)
            {
                throw new Exception("Деление на ноль");
            }
            var x = Convert.ToInt32(a) / Convert.ToInt32(b);
            Debug.Log($"{a} / {b} = {x}");
        }
    }
}
```





Если пользователь введет не число, а строку или некорректные символы, то программа выдаст ошибку. С одной стороны, здесь как раз та ситуация, когда можно применить блок **try catch**, чтобы обработать возможную ошибку. Но оптимально было бы проверить допустимость преобразования:

```
using System;
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            Division("5", "0");
        }

        private void Division(string inputA, string inputB)
        {
            if (!Int32.TryParse(inputA, out var a)) // С версии C# 6.0 мы можем
объявлять переменные прямо при передаче в параметр
            {
                Debug.Log($"Плохое число {inputA}");
            }
            else if (!Int32.TryParse(inputB, out var b))
            {
                Debug.Log($"Плохое число {inputB}");
```

```
            }
            else if(b == 0)
            {
                Debug.Log("Деление на ноль");
            }
            else
            {
                var x = a / b;
                Debug.Log($"{a} / {b} = {x}");
            }
        }
    }
}
```

С версии C# 6.0 мы можем объявлять переменные прямо при передаче в параметр!

Метод **Int32.TryParse()** возвращает **true**, если преобразование можно осуществить, и **false** – если нельзя. При допустимости преобразования переменная **x** будет содержать введенное число. Так, не используя **try catch**, можно обработать возможную исключительную ситуацию. С точки зрения производительности использование блоков **try catch** более накладно, чем применение условных конструкций. Поэтому по возможности вместо **try catch** лучше использовать условные конструкции на проверку исключительных ситуаций.

В C# 6.0 (Visual Studio 2015) была добавлена такая функциональность, как фильтры исключений. Они позволяют обрабатывать исключения в зависимости от определенных условий:

```
int x = 1;
int y = 0;
try
{
    int result = x / y;
}
catch (Exception ex) when (y == 0)
{
    Debug.Log("y не должен быть равен 0");
}
catch (Exception ex)
{
    Debug.Log(ex.Message);
}
```

Генерация собственных исключений

До сих пор мы рассматривали исключения, которые генерирует среда, но сгенерировать исключение может и сам программист. Для этого необходимо воспользоваться оператором **throw**, указав параметры, определяющие вид исключения. Параметром должен быть объект, порожденный от стандартного класса **System.Exception**. Этот объект используется для передачи обработчику информации об исключении:

```
using System;
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
```

```
    {
        private void Start()
        {
            try
            {
                int x = int.Parse("-3");
                if (x < 0) throw new Exception();
                Debug.Log("ok");
            }
            catch
            {
                Debug.Log("введено недопустимое значение");
            }
        }
    }
}
```

Создание собственных исключений:

```
using System;

namespace Geekbrains
{
    public sealed class MyException : Exception
    {
        public int Value { get;}
        public MyException(string message, int val) : base(message)
        {
            Value = val;
        }
    }
}
```

```
using UnityEngine;

namespace Geekbrains
{
    public class Test : MonoBehaviour
    {
        private void Start()
        {
            try
            {
                int x = int.Parse("-3");
                if (x < 0) throw new MyException("введено недопустимое значение", x);
                Debug.Log("ok");
            }
            catch (MyException e)
            {
                Debug.Log($"{e.Message} {e.Value}");
            }
        }
    }
}
```

Советы по работе с исключениями

Несколько общих рекомендаций по обработке исключений:

- Платформа **.NET Framework** активно использует механизм исключений для уведомлений об ошибках и их обработки. Поступайте так же.
- Тем не менее, исключения предназначены для индикации исключительных ситуаций, а не для контроля за ходом выполнения программы. Если объект не может принимать значение **null**, выполняйте простую проверку сравнением, не перекладывая работу на исключение. То же самое относится к делению на ноль и ко многим другим простым ошибкам.
- Исключения стоит применять лишь в крайних ситуациях еще и потому, что они расходуют много памяти и времени.
- Исключения должны содержать максимум полезной информации, помогающей в диагностике и решении проблемы (с учетом предостережений, приведенных ниже).
- Не показывайте необработанные исключения пользователю. Их следует регистрировать в журнале, чтобы разработчики смогли впоследствии устранить проблему.
- Будьте осторожны в раскрытии информации. Помните, что злонамеренные пользователи могут извлечь из исключений информацию о том, как работает программа и какие уязвимости она имеет.
- Не перехватывайте корневой объект всех исключений **system.Exception**. Он поглотит все ошибки, которые необходимо проанализировать и исправить. Это исключение хорошо перехватывать в целях регистрации, если вы собираетесь возбудить его повторно.
- Помещайте исключения низкого уровня в свои исключения, чтобы скрыть детали реализации. Например, если у вас есть коллекция, реализованная с помощью List<T>, имеет смысл скрыть исключение ArgumentOutOfRangeException внутри исключения MyComponentException

Практическое задание

1. По условию, где программа может вести себя неправильно, из-за ошибки гейм и левл дизайнеров, пробросить исключение.
2. Расписать в текстовом документе зачем нужно отлавливать исключения и привести примеры.
3. Добавить событие для подбора бонусов и ловушек и подписать нескольких слушателей на это событие, например потряхивать камеру.
4. *Задание из пункта 1, только использовать свое исключение
5. *Задание из пункта 3, только использовать свое событие

Дополнительные материалы

1. https://docs.unity3d.com/ru/current/ScriptReference/EventSystems.EventSystem.html

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.