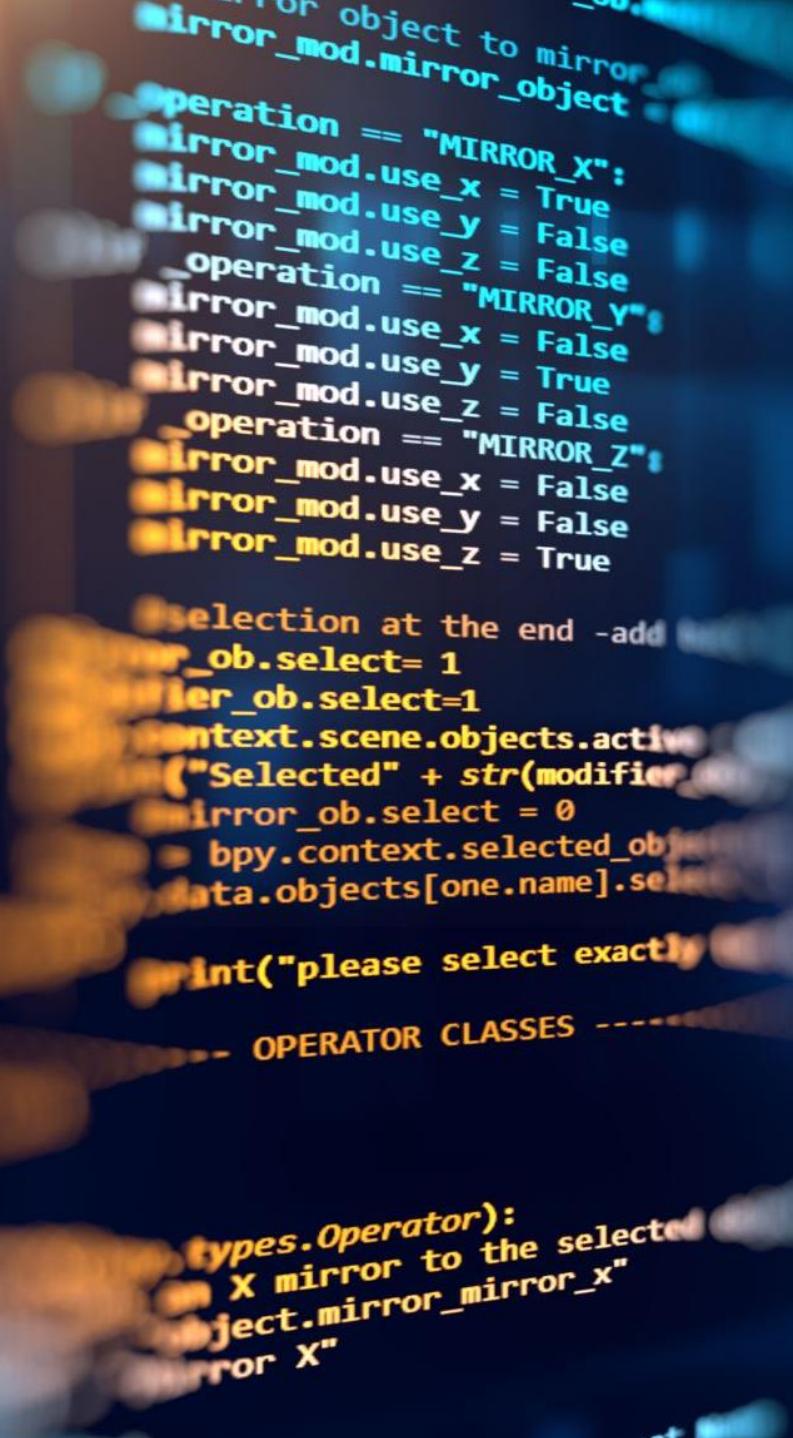


ЯЗЫКИ ПРИКЛАДНОГО ПРОГРАММИРОВАНИЯ

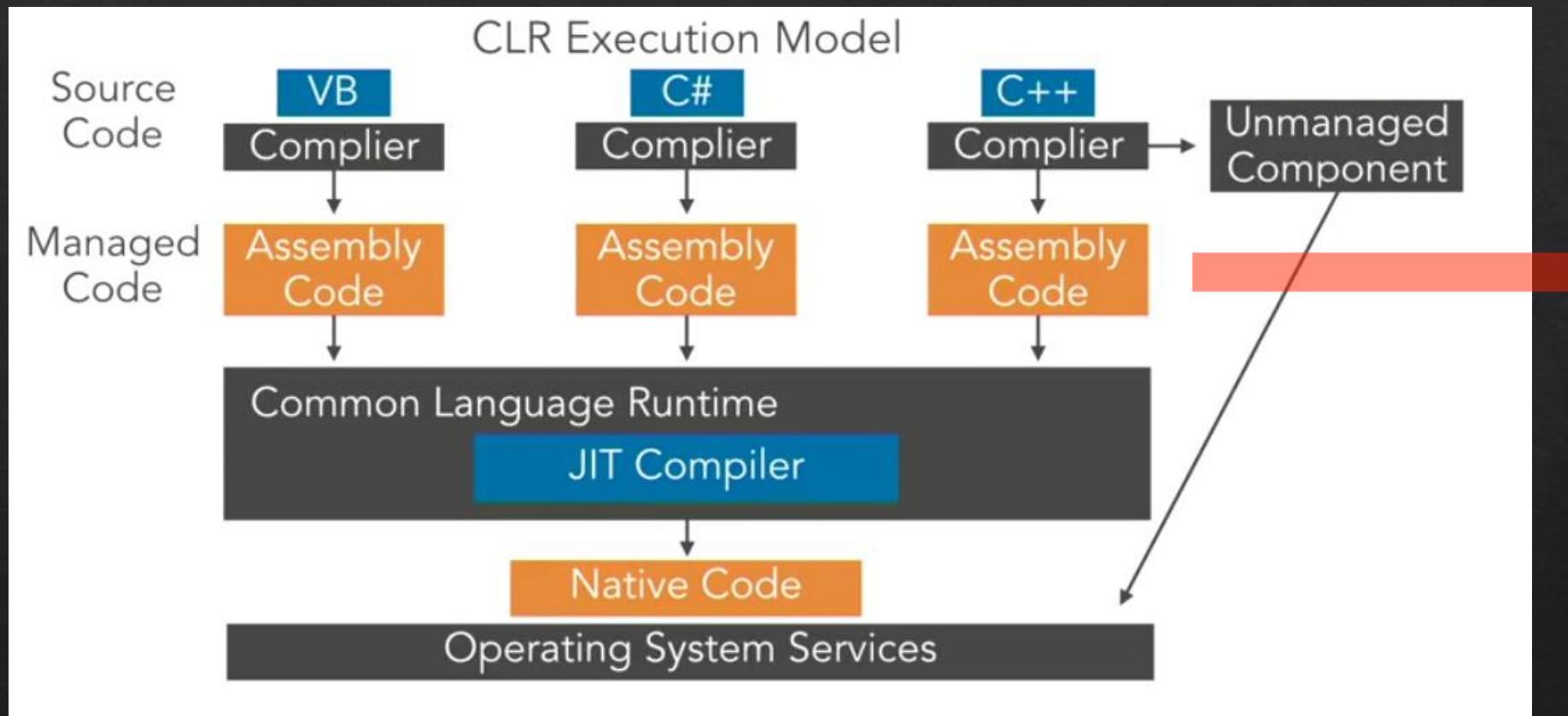
Лекция 8



```
    for object in objects:
        if object == selected_object:
            mirror_mod.mirror_object = object
            mirror_mod.use_x = True
            mirror_mod.use_y = False
            mirror_mod.use_z = False
        else:
            mirror_mod.mirror_object = None
            mirror_mod.use_x = False
            mirror_mod.use_y = False
            mirror_mod.use_z = False

    if len(objects) > 1:
        print("please select exactly one object")
        return {"FINISHED"}  
  
- OPERATOR CLASSES -  
  
class Operator(bpy.types.Operator):
    bl_idname = "object.mirror"
    bl_label = "X mirror to the selected object"
    bl_options = {'REGISTER', 'UNDO'}
```

Неуправляемые ресурсы



Неуправляемые ресурсы

- ❖ Managed – управляемые, очищаются GC (объекты)
- ❖ Unmanaged – неуправляемые, не очищаются GC, требуется «ручная» очистка.
- ❖ Unmanaged: подключения к файлам, базам данных, сетевые подключения и т.д.

- ❖ Очистка реализуется за счет 2-х механизмов:
 - ❖ Деструктора
 - ❖ Реализации стандартного интерфейса *IDisposable*

Деструктор

- ❖ Деструктор не может иметь модификаторов доступа и параметров.
- ❖ Класс может иметь только один деструктор.

```
class Person
{
    public string Name { get; }
    public Person(string name) => Name = name;

    ~Person()
    {
        Console.WriteLine($"'{Name}' has deleted");
    }
}
```

Финализатор

- ❖ Деструктор компилятором будет превращен в финализатор:

```
protected override void Finalize()
{
    try
    {
        // здесь идут инструкции деструктора
    }
    finally
    {
        base.Finalize();
    }
}
```

Деструктор и Финализатор

- ❖ Точное время вызова деструктора не определено.
- ❖ При финализации двух связанных объектов порядок вызова деструкторов не гарантируется.
- ❖ На уровне памяти:
 - ❖ сборщик мусора при размещении объекта в куче , если объект имеет метод *Finalize*, то указатель на него сохраняется в специальной таблице - очередь финализации.
 - ❖ Когда наступает момент сборки этого объекта, сборщик видит, что данный объект должен быть уничтожен, и метод *Finalize*, копируется в еще одну таблицу
 - ❖ Вызван он будет лишь при следующем проходе сборщика мусора.

IDisposable

- ❖ В методе Dispose – очистка неуправляемых ресурсов
- ❖ Выполняется немедленно, при «сборке» объекта

```
public interface IDisposable
{
    void Dispose();
}
```

IDisposable

```
public class Person : IDisposable
{
    public string Name { get; }
    public Person(string name) => Name = name;

    public void Dispose()
    {
        Console.WriteLine($"'{Name}' disposed");
    }
}
```

```
void Exec()
{
    Person? tom = null;
    try
    {
        tom = new Person("Tom");
    }
    finally
    {
        tom?.Dispose();
    }
}
```

Комбинация

- ❖ Finalize оказывает сильное влияние на производительность =>
 - ❖ деструкторы делаем только там, где они нужны
 - ❖ GC.SuppressFinalize – блокируем финализатор после вызова Dispose
- ❖ При создании производных классов от базовых, которые реализуют интерфейс IDisposable, следует также вызывать метод Dispose базового класса

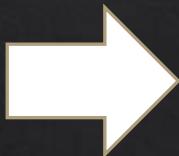
```
public class GoodDisposable : IDisposable
{
    private void ReleaseUnmanagedResources()
    {
        // TODO release unmanaged resources here
    }

    public void Dispose()
    {
        ReleaseUnmanagedResources();
        GC.SuppressFinalize(this);
    }

    ~GoodDisposable()
    {
        ReleaseUnmanagedResources();
    }
}
```

Using

```
void Exec()
{
    Person? tom = null;
    try
    {
        tom = new Person("Tom");
        // some actions
    }
    finally
    {
        tom?.Dispose();
    }
}
```



```
void ExecV1()
{
    using (Person tom = new Person("Tom"))
    {
        // some actions
    }
}

void ExecV2()
{
    using Person tom = new Person("Tom");
    // some actions
}
```

Работа с файлами

- ❖ При помощи статических методов класса File
- ❖ Со всеми видами файлов - FileStream
- ❖ Текстовые – при помощи StreamReader, StreamWriter
- ❖ Бинарные – при помощи BinaryWriter, BinaryReader

Работа с файлами

```
using (StreamReader reader = new StreamReader(path))
{
    string text = reader.ReadToEnd();
    Console.WriteLine(text);
}

using (StreamWriter writer = new StreamWriter(path, false))
{
    writer.WriteLine(text);
}
```

Методы Расширения

Позволяют добавлять новые методы в уже существующие типы без создания нового производного класса.

Эта функциональность бывает особенно полезна, когда необходимо добавить в тип новый метод, но сам тип (класс или структуру) мы изменить не можем, поскольку у нас нет доступа к исходному коду.

```
public static class StringExtension
{
    public static int CharactersCount(this string str, char c)
    {
        var counter = 0;
        for (var i = 0; i < str.Length; i++)
        {
            if (str[i] == c)
                counter++;
        }
        return counter;
    }
}
```

LINQ

LINQ (Language-Integrated Query)

- ❖ язык запросов к источнику данных.
- ❖ Варианты источников данных:
 - ❖ IEnumerable (LINQ to Objects),
 - ❖ DataSet (LINQ to DataSet, ADO .NET),
 - ❖ База данных (LINQ to Entities, EF),
 - ❖ документ XML ...
 - ❖ ...
- ❖ Вне зависимости от типа источника LINQ позволяет применить ко всем один и тот же подход.

Способ применения

- Операторы запросов
- Методы расширений

```
public static void UsingOperators()
{
    string[] people = { "Tom", "Bob", "Sam", "Tim", "Tomas", "Bill" };

    // создаем новый список для результатов
    var selectedPeople = from p in people
        // передаем каждый элемент из people в переменную p
        where p.ToUpper().StartsWith("T") //фильтрация по критерию
        orderby p // упорядочиваем по возрастанию
        select p; // выбираем объект в создаваемую коллекцию

    foreach (var person in selectedPeople)
        Console.WriteLine(person);
}

public static void UsingExtensions()
{
    string[] people = { "Tom", "Bob", "Sam", "Tim", "Tomas", "Bill" };

    var selectedPeople = people
        .Where(p => p.ToUpper().StartsWith("T"))
        .OrderBy(p => p);

    foreach (var person in selectedPeople)
        Console.WriteLine(person);
}
```

Готовимся

```
public class UserAction
{
    public UserAction(string actionText, DateTime date)
    {
        ActionText = actionText;
        Date = date;
    }

    public string ActionText { get; }

    public DateTime Date { get; }
}

string[] people = { "Tom", "Bob", "Sam", "Tim", "Tomas", "Bill" };
```

Where

- ❖ Фильтрация значений коллекции по условию

```
string[] people = { "Tom", "Bob", "Sam", "Tim", "Tomas", "Bill" };

var where = people.Where(x => x.StartsWith("T"));
```

Select

- ❖ Проекция коллекции

```
var hellos = people.Select(x => $"Hello, {x}!");  
var lengths = people.Select(x => x.Length);
```

Select Many

- ❖ Проекция коллекции, объединяющая коллекции

```
int[][] arrays = {
    new[] {1, 2, 3},
    new[] {4},
    new[] {5, 6, 7, 8},
    new[] {12, 14}
};

// Will return { 1, 2, 3, 4, 5, 6, 7, 8, 12, 14 }
var result = arrays.SelectMany(array => array);
```

ANY & All

- ❖ **All**: определяет, все ли элементы коллекции удовлетворяют определенному условию
- ❖ **Any**: определяет, удовлетворяет хотя бы один элемент коллекции определенному условию

```
var haveSam = people.Any(x => x == "Sam");

var haveNoEmptyStrings = people.All(x => !string.IsNullOrWhiteSpace(x));
```

Count

- ❖ Подсчитывает количество элементов коллекции, которые удовлетворяют определенному условию

```
var peoplesCount = people.Count();  
  
var peoplesStartsFromTCount = people.Count(x => x.StartsWith("T"));
```

Order by & order by descending

- ❖ **OrderBy:** упорядочивает элементы по возрастанию
- ❖ **OrderByDescending:** упорядочивает элементы по убыванию

```
UserAction[] actions = new[]
{
    new UserAction("Authorize", DateTime.Now.AddDays(-1)),
    new UserAction("Authorize", DateTime.Now.AddDays(-2)),
    new UserAction("Authorize", DateTime.Now.AddDays(-3)),

    new UserAction("Update name", DateTime.Now.AddDays(-2).AddHours(-2)),
    new UserAction("Update email", DateTime.Now.AddDays(-2)),
    new UserAction("Confirm email", DateTime.Now.AddDays(-2)),
    new UserAction("Fill balance", DateTime.Now.AddDays(-1)),
};

var ordered = actions.OrderBy(x => x.Date);

var orderedByDesc = actions.OrderByDescending(x => x.Date);
```

First & first or default

- ❖ **First**: выбирает первый элемент коллекции (опционально – по условию)
- ❖ **FirstOrDefault**: выбирает первый элемент коллекции (опционально – по условию) или возвращает значение по умолчанию

```
var noActions = new List<UserAction>();

var firstName = people.First();
var firstNameStartsFromT = people.First(x => x.StartsWith("T"));
var firstUpdateAction = actions.First();
actions.First(); // exception

var defaultAction = actions.FirstOrDefault(); // null
```

Last & last or default

- ❖ Как First, только Last =)

single & single or default

- ❖ **Single**: выбирает единственный элемент коллекции, если коллекция содержит больше или меньше одного элемента, то генерируется исключение
- ❖ **SingleOrDefault**: выбирает единственный элемент коллекции. Если коллекция пуста, возвращает значение по умолчанию. Если в коллекции больше одного элемента, генерирует исключение

```
var singleConfirm = actions.Single(x => x.ActionText.Contains("Confirm"));
var singleAuthorize = actions.Single(x => x.ActionText == "Authorize"); // exception
var single = actions.Single(); // exception

defaultAction = actions.SingleOrDefault(); // exception
defaultAction = actions.SingleOrDefault(x => string.IsNullOrWhiteSpace(x.ActionText)); null
```

Sum, Average, min, max

```
var numbers = new List<int>() { 1, 3, 2, 1};

var numbersSum = numbers.Sum(); // 7
var min = numbers.Min(); // 1
var max = numbers.Max(); // 3
var avg = numbers.Average(); // 7 / 4
```

Aggregate

- ❖ **Aggregate**: применяет к элементам последовательности агрегатную функцию, которая сводит их к одному объекту

```
var names = people.Aggregate("Names:", (first, next) => $"{first} {next}");
// Names: Tom, Bob, Sam, Tim, Tomas, Bill
```

Distinct

- ❖ **Distinct**: удаляет дублирующийся элементы из коллекции

```
var distinctNumbers = numbers.Distinct(); // 1, 2, 3
```

TAKE & SKIP

- ❖ **Take**: выбирает определенное количество элементов
- ❖ **Skip**: пропускает определенное количество элементов

```
var firstTwoActions = actions.Take(2);

var skipTwoActions = actions.Skip(2);

var takeTwoActionsInSecondPage = actions.Skip(2).Take(2);
```

Group by & ToLookup

- ❖ **GroupBy**: группирует элементы по ключу
- ❖ **ToLookup**: группирует элементы по ключу, при этом все элементы добавляются в словарь

```
var actionsByDate = actions.GroupBy(x => x.Date);

var actionsByDateDictionary = actions.ToLookup(x => x.Date);
```

Except, Union, Intersect

