Платформа .Net C#

Полезные классы из System

- AppDomain- среда, в которой выполняются приложения.
- Convert- преобразует значение одного базового типа данных к другому.
- Enviroment- сведения о текущей среде и платформе.
- GC- Управляет системным сборщиком мусора.
- WeakReference- слабая ссылка, которая указывает на объект, но позволяет удалять его сборщику мусора.

Зачем нужны исключения?

- Решение проблем с кодом вида:

  return -1;

- Возможность реакции на системные ошибки, например: деление на ноль.
- Структурирование кода приложения.
- Решение или фиксация в логах проблемных ситуаций.

Базовый синтаксис

```
try {
    Funk1();
    Funk2();
}
catch (Exception1 ex) { . . . }
catch (Exception2 ex) { . . . }
```

Важно: - Любое исключение наследует от класса Exception;

- Блоков **catch** один или больше;
- Возможен универсальный блок: **catch(Exception** ex){ }
- Не все исключения нам предложат обработать.

Полная версия синтаксиса

```
try {
    Funk1();
    Funk2();
}
catch (Exception1 ex) { . . . }
catch (Exception   ex) { . . . }
finally {
    // код освобождения ресурсов
 }
```

- Возможна сокращённая версия – без **catch**.

Выброс исключений

- Виртуальной машиной CLR.
- Библиотечным методом (см. документацию)
- **throw new** Exception();
- **throw**; // только в блоке **catch**

Обсудить:

- **catch(Exception ex) { }**
- **if (obj==null) throw new NullReferenceException();**
- int? n= **Customer?.Name?.Length**;

Создание своих классов исключений

```
[Serializable()]
public class MyException : Exception
 {
    public MyException () : base() { }
    public MyException (string message)  :    base (message) { }
    public MyException (string message,  Exception inner) :
                                base (message, inner) { }
    protected MyException ( SerializationInfo info,
                        StreamingContext context)  { }
}
```

Контроль переполнения при целочисленных операциях

- Для выражения:

  checked(выражение);

  unchecked(выражение);
- Для блока кода:

  checked { }

  unchecked { }
- Для всего приложения (см. свойства проекта)

И какой результат?

- Проверка включена:
  - Для константного выражения – ошибка при компиляции приложения.
  - Не константное выражение – OverflowException.
- Проверка выключена:
  - Усечение результата, потеря старших битов.

Объявление интерфейса

```
public interface IWork {
    void DoWork();
}
public interface IWorkGeom :  IWork {
    double SolveGeom();
}
```

Важно: - Это контракт, т.е. должен оставаться неизменным;

- Нет реализации;
- Может включать методы, свойства, события и индексаторы;
- Все элементы по умолчанию **public**;
- Это ссылочный тип данных.

Реализация интерфейсов

```
class Star : IWorkGeom {
    public void DoWork() { }
    public double SolveGeom() {return 0;}
    public void Move() { }
}
. . .
Star star = new Star();
 IWork work = star;
    star.
```

DoWork
SolveGeom
Move

work.

DoWork

Явная реализация интерфейса

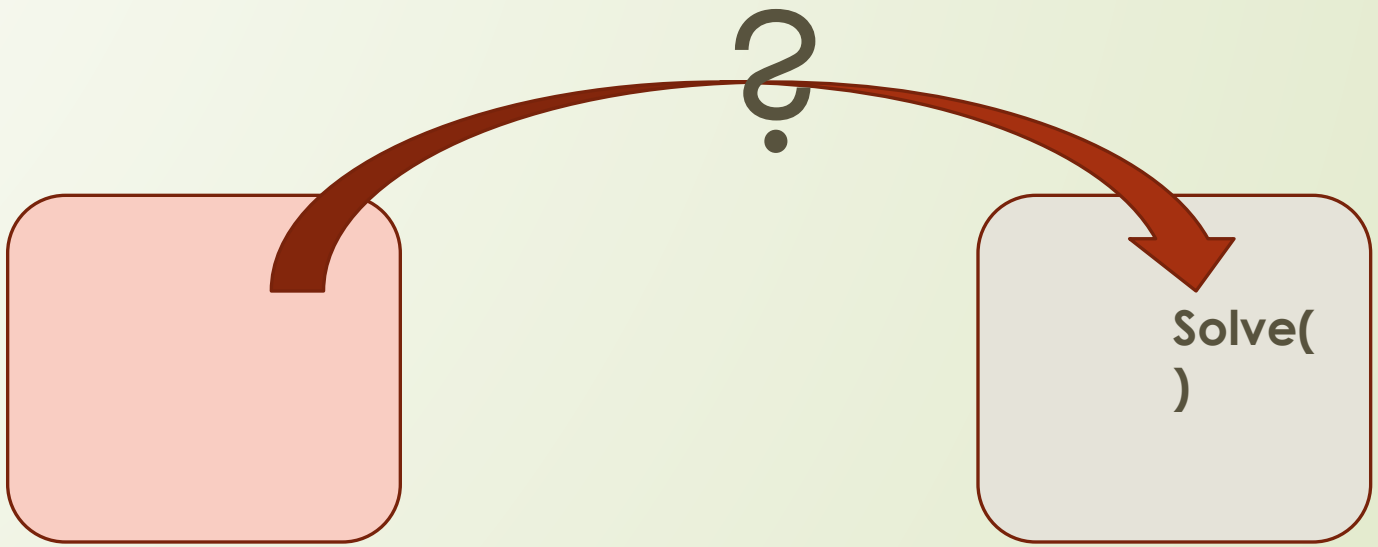
```
public interface IWork {   void Paint(); }
public interface IDraw {   void Paint(); }

class Star : object,   IWork, IDraw {
      void IWork.Paint() { }
      void IDraw.Paint() { }
}
```

Важно: - Доступ к методу Paint() возможен только по интерфейсной ссылке;

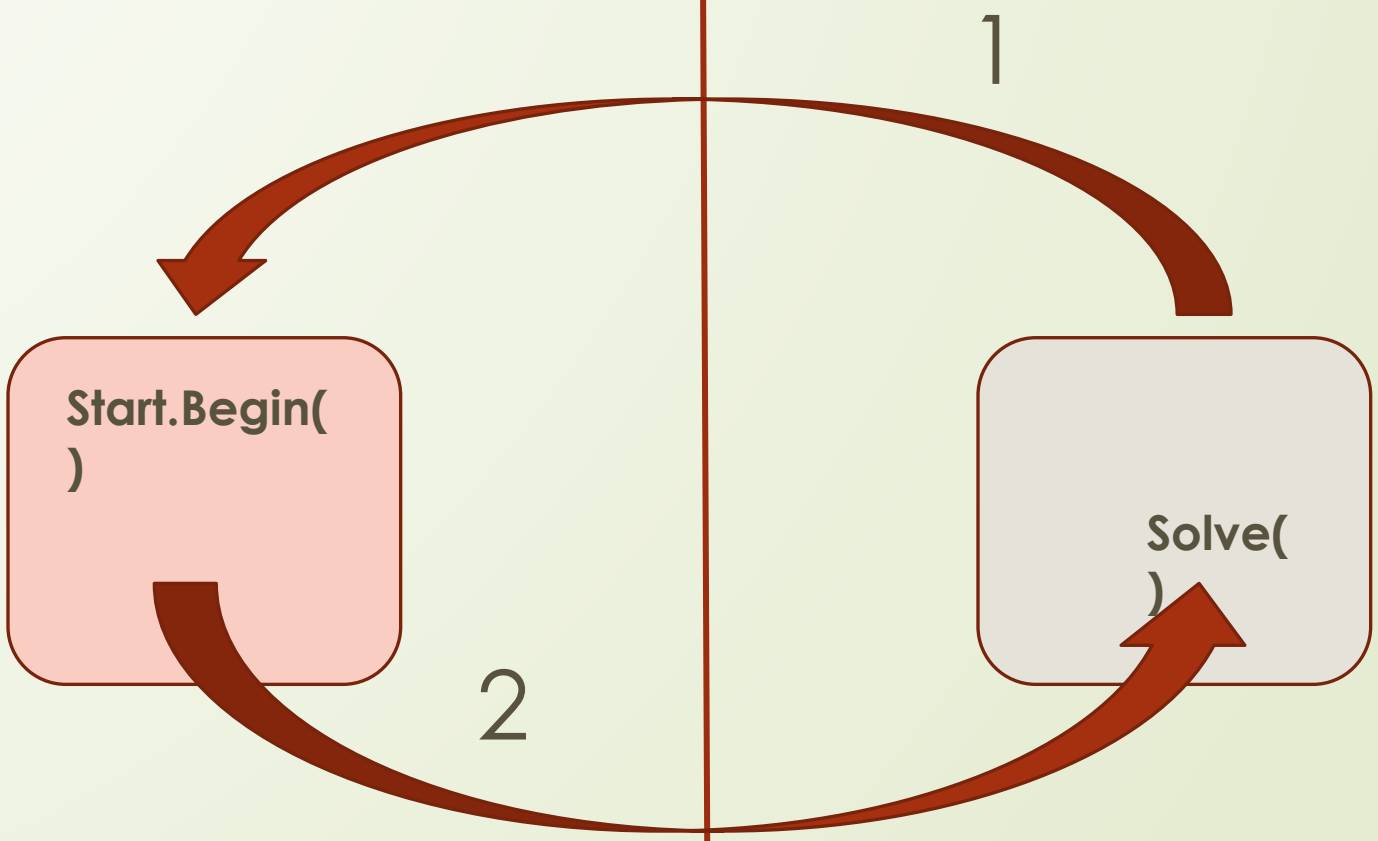
- Нет модификатора доступа;
- Нет модификаторов **abstract**, **virtual**, **override**, или **static**;

Интерфейсы, как способ взаимодействия объектов.



```
public class Start {
    static void
     Begin(IWork work) {
     work.Solve();
    }
}
```

```
public interface IWork {
    void Solve();
}
```



Абстрактный класс и интерфейсы

```
public interface IDraw   {
        void Paint();
}
abstract  class  Star :  IDraw {
        public abstract void Paint();
        . . .
}
```

Структурный тип и интерфейсы

```
public interface IMath   {
      void AddOne();
}
struct  Star :  IMath {
     public  int  num;
     public  void AddOne() { ++num; }
}
        Star  star = new Star();          // num=0
        star.AddOne();              // num=1
        IMath math=star;
        math.AddOne();
        Console.WriteLine(star.num);          // Что будет распечатано?
```

Стандартный итератор .Net

- Все коллекции реализуют интерфейс:

```
public interface IEnumerable {
    IEnumerator GetEnumerator();
}
```

- Который и даёт доступ к итератору:

```
public interface IEnumerator {
    object Current { get; }
    bool MoveNext();
    void Reset();
}
```

Работа с файлами и каталогами

- File - статические методы для создания, копирования, удаления и перемещения файлов.
- FileInfo - методы экземпляра для создания, копирования, удаления и перемещения файлов.
- Directory - статические методы для создания, перемещения и перечисления файлов в каталогах.
- DirectoryInfo - методы экземпляра для создания, перемещения и перечисления файлов в каталогах.
- Path - методы и свойства для обработки имён каталогов кроссплатформенным способом.

- Добавление текста в файл:

```
string path = @"C:\Users\student\books.txt";
if (File.Exists(path))
    File.AppendAllText(path, “some text \r\n new line");
```

- Поиск по дереву каталогов:

```
string path = @"C:\Users\student\Desktop";
string[] books = Directory.GetFiles(path, "*.pdf",
SearchOption.AllDirectories);
foreach (string s in books)      Console.WriteLine(s);
```

Потоки байтов

- FileStream — для чтения и записи в файл.
- MemoryStream — для чтения и записи в память.
- BufferedStream — для повышения быстродействия операций чтения и записи.
- NetworkStream — для чтения и записи на сетевые сокеты.
- PipeStream — для чтения и записи в анонимные и именованные каналы.
- CryptoStream — для связи потоков данных с криптографическими алгоритмами.

- Чтение байтов из файла:

```
string path = @"C:\Users\student\books.txt";
FileStream fs = File.OpenRead(path);
// FileStream fs = new FileStream(path, FileMode.Open, FileAccess.Read);
int  ch;
while( (ch = fs.ReadByte())  !=  -1) Console.Write(char.ConvertFromUtf32(ch));

fs.Close();                    // вызывает Dispose(true)
fs.Dispose();                  // рекомендуется вызывать именно его
```

Потоки символов

- BinaryReader и BinaryWriter - для чтения и записи простых типов данных, например: double.

- TextReader и TextWriter - абстрактные базовые классы. Их наследники:
  - StreamReader и StreamWriter - для чтения и записи текстов с учётом кодировки символов.
  - StringReader и StringWriter - для чтения и записи символов в строки или из строк.

Чтение текстового файла:

```
string  line;
StreamReader  tr = null;
 string path = @"C:\Users\student\books.txt";
 try
 {
     tr = new StreamReader(path, Encoding.GetEncoding(1251));
     while ((line = tr.ReadLine()) != null)    Console.WriteLine(line);
 }
 finally {
if (tr!=null)       tr.Dispose();
}
```

Работа с архивами

- ZipArchive — набор сжатых файлов в формате ZIP.
- ZipArchiveEntry — сжатый файла в архиве.
- ZipFile — статические методы для работы с архивом.
- ZipFileExtensions — расширяющие методы для работы с архивом.

- DeflateStream — для сжатия и распаковки потоков с помощью алгоритма Deflate.
- GZipStream — для сжатия и распаковки потоков в формате gzip.

Извлечение всех файлов из архива ZIP:

```
string zipPath = @"C:\Users\student\Desktop\java.zip";
string extractPath = @"C:\Users\student\Desktop";

using (ZipArchive archive =
  ZipFile.Open(zipPath, ZipArchiveMode.Read))
{
        archive.ExtractToDirectory(extractPath);
}
```