

Обработка на грешки

- Грешка – нарушение на допусканията на един програмен интерфейс (програмна логика)
- Кога се нарушават допусканията на програмната логика?
 - При грешна логиката
 - При грешно подадени входни данни данни (от потребителя или прогр.)
 - При грешна последователност от действия (от потребителя или прогр.)

Пр: грешка в логиката на п-мата

```
int[] array = new int[5];  
  
for (int i = 0; i < 10; i++)  
{  
    array[i] = i; // array.Length == 5  
}
```

```
int[] array = null;  
  
for (int i = 0; i < 10; i++)  
{  
    array[i] = i; // array == null  
}
```

Пр: грешно подадени вх. данни от потребителя

```
Console.Write("Enter num:");  
string numText = Console.ReadLine(); //numText == "aaa"  
  
int num = int.Parse(numText);
```

Обработка на грешки

- Чрез директна проверка и диалог
- Чрез връщана стойност на функция
- Чрез изключения

Директна проверка и диалог

```
Console.Write("Write name:");  
string name = Console.ReadLine();  
  
if (name.Length == 0)  
{  
    Console.WriteLine("Empty name not allowed!");  
    return;  
}  
  
...
```

Директна проверка и диалог

- Удачна е само на места, където е предвиден диалог с потребителя

Връщана стойност на функция

```
enum StoreNameResult
{
    Success = 0,
    EmptyNameError = 1,
    NameTooShortError = 2,
}
```

```
public static StoreNameResult StoreName(string name)
{
    if (name.Length == 0)
        return StoreNameResult.EmptyNameError;

    if(name.Length < 2)
        return StoreNameResult.NameTooShortError;

    ...

    return StoreNameResult.Success;
}
```

Връщана стойност на функция

- Недостатък е ограниченото количество информация за грешките, както и ограничаването на програмните възможности (връщания резултат от функциите винаги е зает)

Исключения

- Структурно ориентиран подход за обработка на грешки
- В .NET се реализира, чрез обекти от тип Exception (или негови производни типове)

Исключения

```
public int StoreName(string name)
{
    if (name.Length == 0)
        throw new Exception("Name is empty");

    if(name.Length < 2)
        throw new Exception("Name is too short");

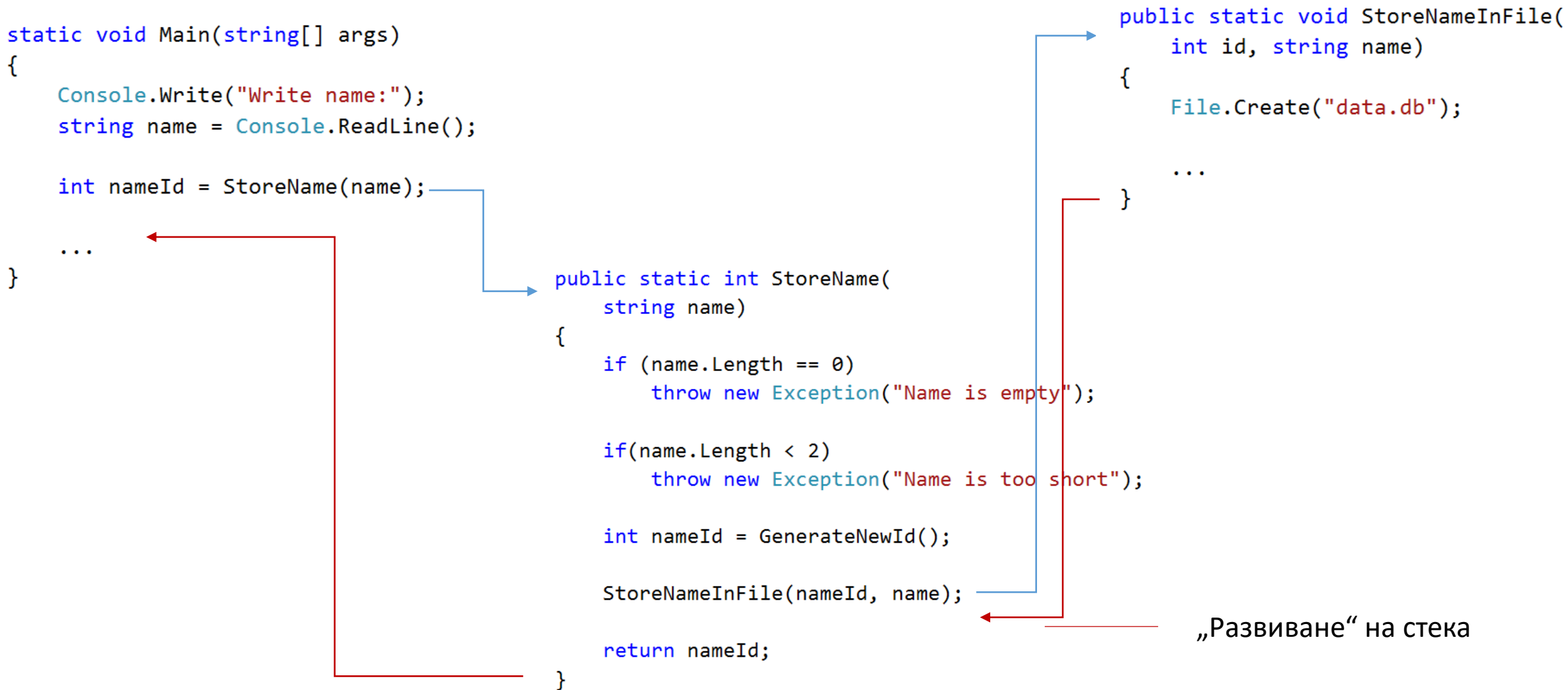
    ...

    return nameId;
}
```

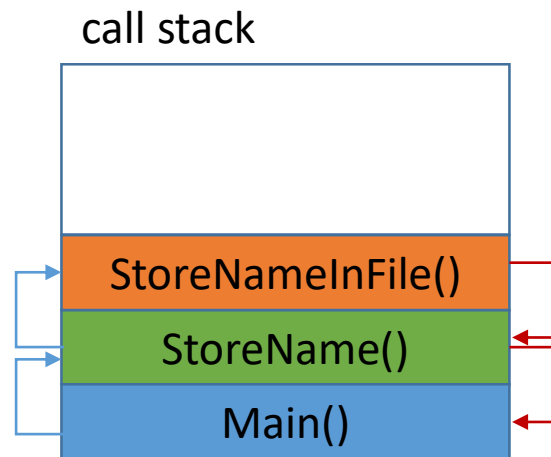
Exception

- Тип за представяне на грешка
- Притежава свойство Message от текстов тип, в което може да се запише описание на грешката
- Подаване на грешки се извършва, чрез ключовата дума throw

Стек на извикванията



Стек на извикванията



„Развиване“ на стека

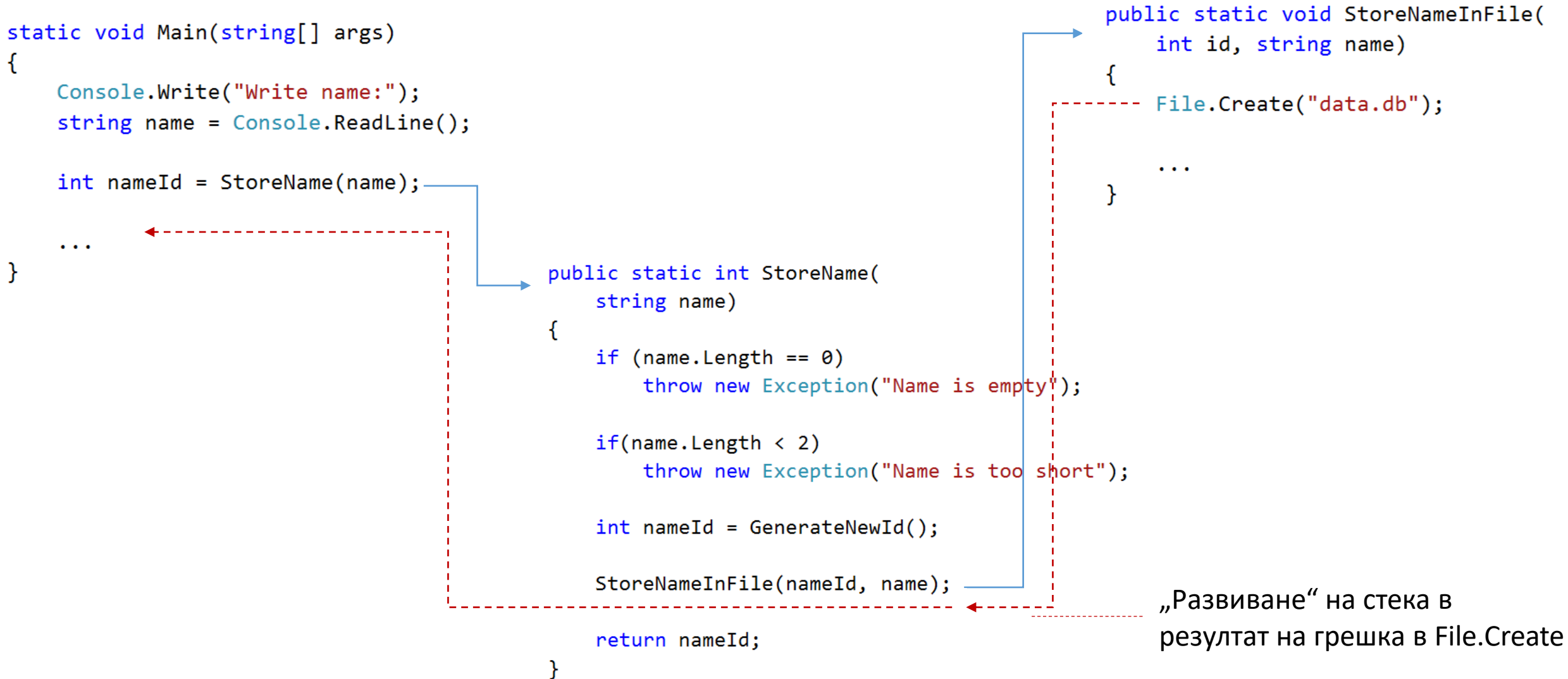
„Развиване“ на стека

- Настъпва при приключване на методите, при което управлението се предава назад по веригата на извикванията

Прихващане на изключение

- Предоставя възможност на програмиста да изпълни код в случай на възникнала грешка (хвърлено изключение)
- Възникването на грешка води до „развиване“ на стека **до** прихващане на грешката

„Развиване“ на стека в резултат на грешка



Прихващане на изключение

```
try
{
}
catch
{
}
```

```
try
{
}
catch(Exception e)
{
}
```

```
try
{
}
finally
{
}
```

```
try
{
}
catch
{
}
finally
{
}
```

Стек на извикванията

```
static void Main(string[] args)
{
    Console.Write("Write name:");
    string name = Console.ReadLine();

    int nameId = StoreName(name);
    ...
}
```

```
public static int StoreName(
    string name)
{
    if (name.Length == 0)
        throw new Exception("Name is empty");

    if(name.Length < 2)
        throw new Exception("Name is too short");

    int nameId = GenerateNewId();

    StoreNameInFile(nameId, name);

    return nameId;
}
```

```
public static void StoreNameInFile(
    int id, string name)
{
    try
    {
        File.Create("data.db");
    }
    catch
    {
    }
    ...
}
```

„Развиване“ на стека

„Развиване“ на стека в резултат на грешка

```
static void Main(string[] args)
{
    Console.WriteLine("Write name:");
    string name = Console.ReadLine();

    int nameId = StoreName(name);
    ...
}

public static int StoreName(
    string name)
{
    if (name.Length == 0)
        throw new Exception("Name is empty");

    if(name.Length < 2)
        throw new Exception("Name is too short");

    int nameId = GenerateNewId();

    try
    {
        StoreNameInFile(nameId, name);
    }
    catch
    {
    }

    return nameId;
}

public static void StoreNameInFile(
    int id, string name)
{
    File.Create("data.db");
    ...
}
```

„Развиване“ на стека в резултат на грешка в File.Create

try блок

- Блок код съдържащ логика, която може да породии грешка или код, след който задължително трябва да се изпълни друг код (най-често за освобождаване на ресурси).

finally блок

- Блок код изпълняван след try блок, който се изпълнява задължително. Най-често се използва за освобождаване на ресурси.

catch блок

- Блок код изпълняван в случай на възникване на грешка. Възможно е да има няколко такива блока един след друг, прихващащи различни типове грешки.

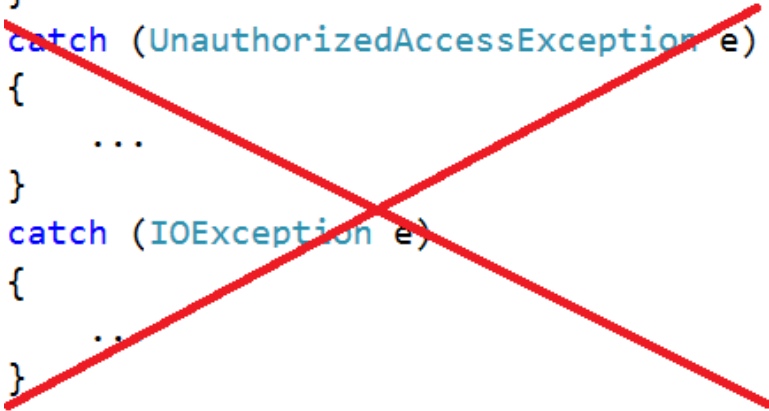
try
catch
finally

```
FileStream stream = null;

try
{
    stream = File.Create("data.db");
}
catch (UnauthorizedAccessException e)
{
    ...
}
catch (IOException e)
{
    ...
}
catch (Exception e)
{
    ...
}
finally
{
    if (stream != null)
    {
        stream.Close();
    }
}
```

```
FileStream stream = null;

try
{
    stream = File.Create("data.db");
}
catch (Exception e)
{
    ...
}
catch (UnauthorizedAccessException e)
{
    ...
}
catch (IOException e)
{
    ...
}
finally
{
    if (stream != null)
    {
        stream.Close();
    }
}
```



?

```
static int FuncaDoodleDoo()
{
    int num = 0;
    for (int i = 0; i < 10; i++ )
    {
        try
        {
            if (i == 1) continue;
            if (i == 2) throw new Exception();
            if (i == 3) break;
        }
        catch
        {
            num--;
        }
        finally
        {
            num++;
        }

        num++;
    }

    return num; //num = ?
}
```


Пораждане на изключения

- Машинни
- Потребителски

Производительность при обработке на грешки

```
Console.Write("Enter number:");
var numText = Console.ReadLine();

try
{
    int num = int.Parse(numText);
}
catch (FormatException e)
{
    Console.Write("Number not valid!");
    return;
}
```

```
Console.Write("Enter number:");
var numText = Console.ReadLine();

int num;
if (!int.TryParse(numText, out num))
{
    Console.Write("Number not valid!");
    return;
}
```

Атрибути

- Декларативни типове (наследници на класа Attribute) служещи за прикрепяне на описателна информация към декларациите в кода
- Могат да се извличат и използват от различни инструменти
- В .NET има множество стандартни атрибути, но могат да се създават и потребителски

Атрибути

```
[Serializable]
```

```
public class Rectangle  
{  
    ...  
}
```

```
[Serializable()]
```

```
public class Rectangle  
{  
    ...  
}
```

```
[SerializableAttribute()]
```

```
public class Rectangle  
{  
    ...  
}
```

Serializable

```
using System.Runtime.InteropServices;

namespace System
{
    ...public sealed class SerializableAttribute : Attribute
    {
        ...public SerializableAttribute();
    }
}
```

Приложение на атрибутите

- Могат се „залепват“ към различни декларации, като това се оказва с атрибута `AttributeUsage` приложен към типа на съответния атрибут

Приложение на атрибутите

```
[AttributeUsage(  
    AttributeTargets.Class |  
    AttributeTargets.Struct |  
    AttributeTargets.Enum |  
    AttributeTargets.Delegate,  
    Inherited = false)]  
[ComVisible(true)]  
public sealed class SerializableAttribute : Attribute  
{  
    ...public SerializableAttribute();  
}
```

Сериализация

- Процес на преобразуване на обект(и) в поток от байтове
- Потока може да е двоичен или текстов

Десериализация

- Процес на преобразуване на поток от байтове в обект(и)

Сериализация

```
IFormatter formatter = new BinaryFormatter();  
  
using (Stream stream = new FileStream("data.db", FileMode.Create, FileAccess.Write))  
{  
    formatter.Serialize(stream, _rectangles);  
}
```

Десериализация

```
IFormatter formatter = new BinaryFormatter();  
  
using (var stream = new FileStream("data.db", FileMode.Open, FileAccess.Read))  
{  
    _rectangles = (List<Rectangle>)formatter.Deserialize(stream);  
}
```

Изисквания за SerializableAttribute

- Полетата на обекта трябва да са структурни типове или класове, които също са сериализуеми.
- Ако някое поле е обект, който не може да се сериализира ще бъде генерирана грешка (exception) от тип SerializationException
- За по детайлно определяне кои полета да се сериализират трябва да се имплементира интерфейса ISerializable
- Поле може да се изключи от сериализация, чрез прилагане на атрибута NonSerialized

NonSerialized

```
[Serializable]

public class Rectangle
{
    public Point Position { get; set; }

    public int Width { get; set; }

    public int Height { get; set; }

    public Color Color { get; set; }

    [NonSerialized]

    public Int32 Order { get; set; }

    ...
}
```

Да се модифицира програмата

- Фигурите да се записват при затваряне на програмата (събитие `FormClosing`)
- Фигурите да се прочитат при стартиране на програмата (събитие `Load`)
- Да се прочита и избраната при затваряне фигура

Да се модифицира програмата

```
private void FormMain_FormClosing(object sender, FormClosingEventArgs e)
{
    IFormatter formatter = new BinaryFormatter();

    using (Stream stream = new FileStream("data.db", FileMode.Create, FileAccess.Write))
    {
        formatter.Serialize(stream, _rectangles);
    }
}
```

Да се модифицира програмата

```
private void FormMain_Load(object sender, EventArgs e)
{
    if (!File.Exists("data.db"))
    {
        return;
    }

    IFormatter formatter = new BinaryFormatter();

    using (var stream = new FileStream("data.db", FileMode.Open, FileAccess.Read))
    {
        _rectangles = (List<Rectangle>)formatter.Deserialize(stream);
    }

    _selectedRectangle = _rectangles
        .Where(c => c.Color == Color.Red)
        .SingleOrDefault();
}
```


Библиотеки

- Библиотеката е асембли
- Няма входна точка (метод `main`)
- Обикновено има разширение `.dll` и Windows няма програма по подразбиране за този тип файл
- Може да се реферира в .NET проекти публичните типове декларирани в нея да се използват

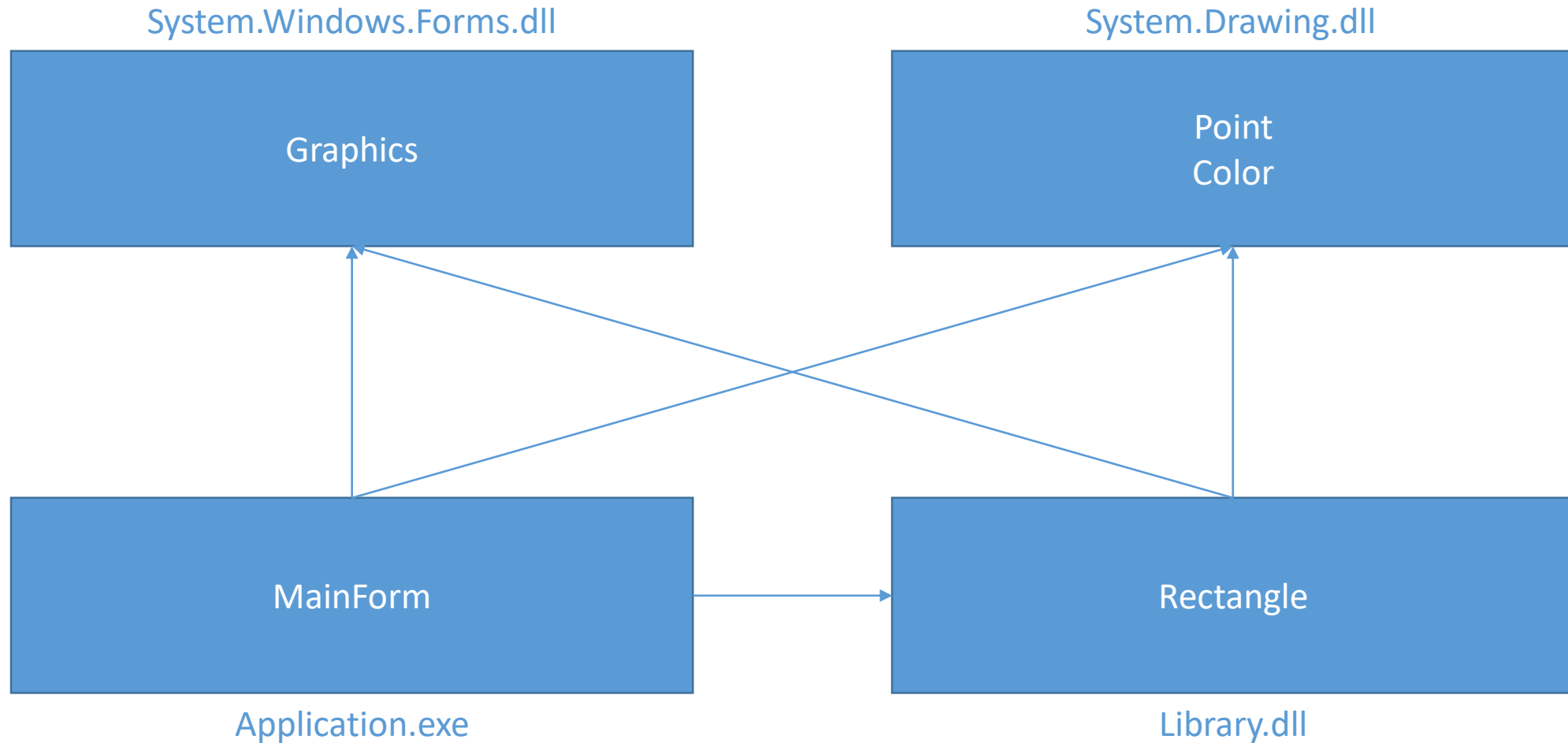
Кога да използваме библиотеки?

- Когато едни и същи типове се използват многократно в различни проекти. Обемът на кода намалява, промени в логиката изискват промяна само на едно място
- Когато има относително независими типове
- Библиотеките могат да имат версии, което предотвратява възможни конфликти между приложения ползващи споделени библиотеки с различни версии (проблем известен като „Dll Hell“)

Как да се модифицира програмата?

- Коя част от логиката на програмата е удачна за прехвърляне в библиотека?

Зависимости между библиотеками

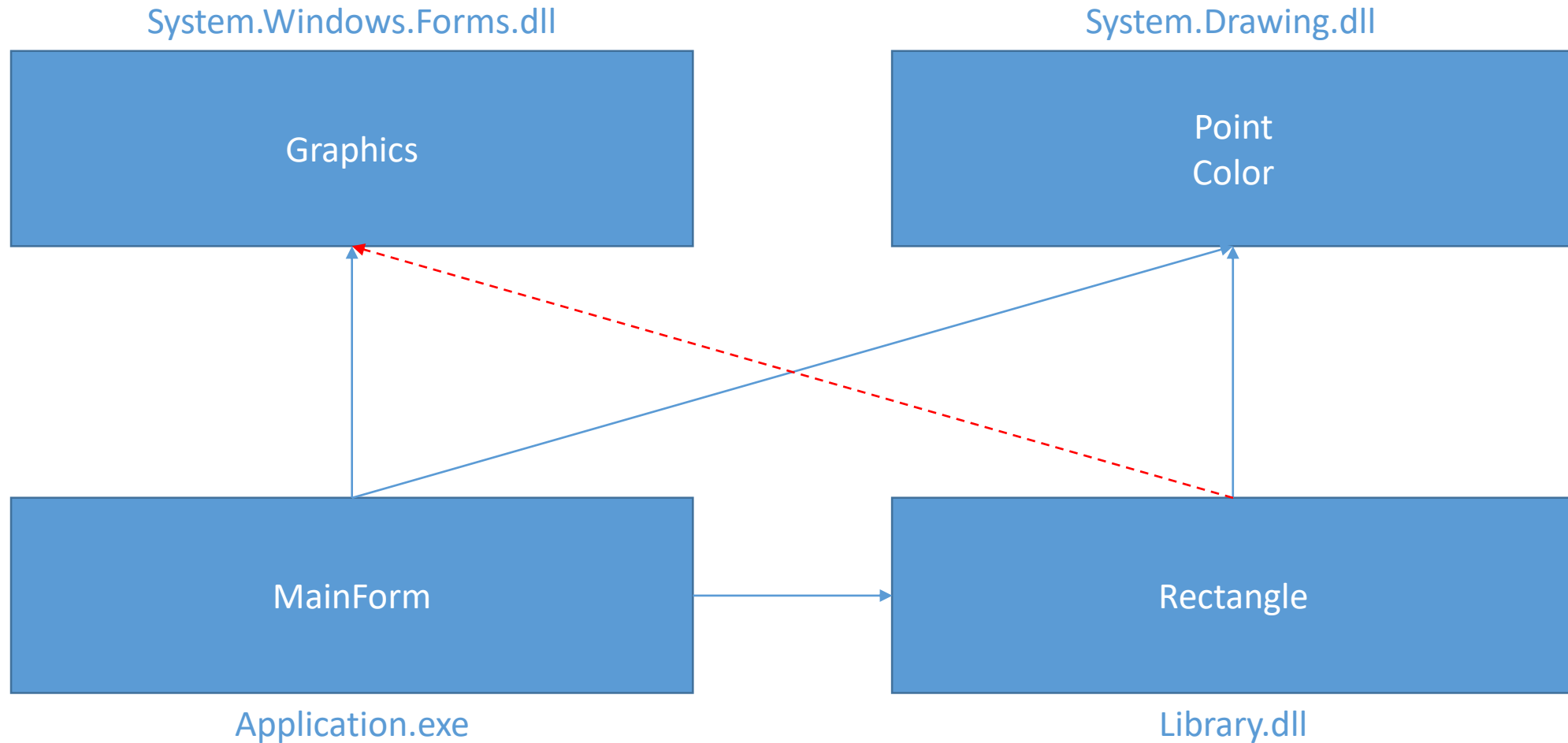


Може ли някоя зависимост да се премахне?

Може ли някоя зависимост да се премахне?

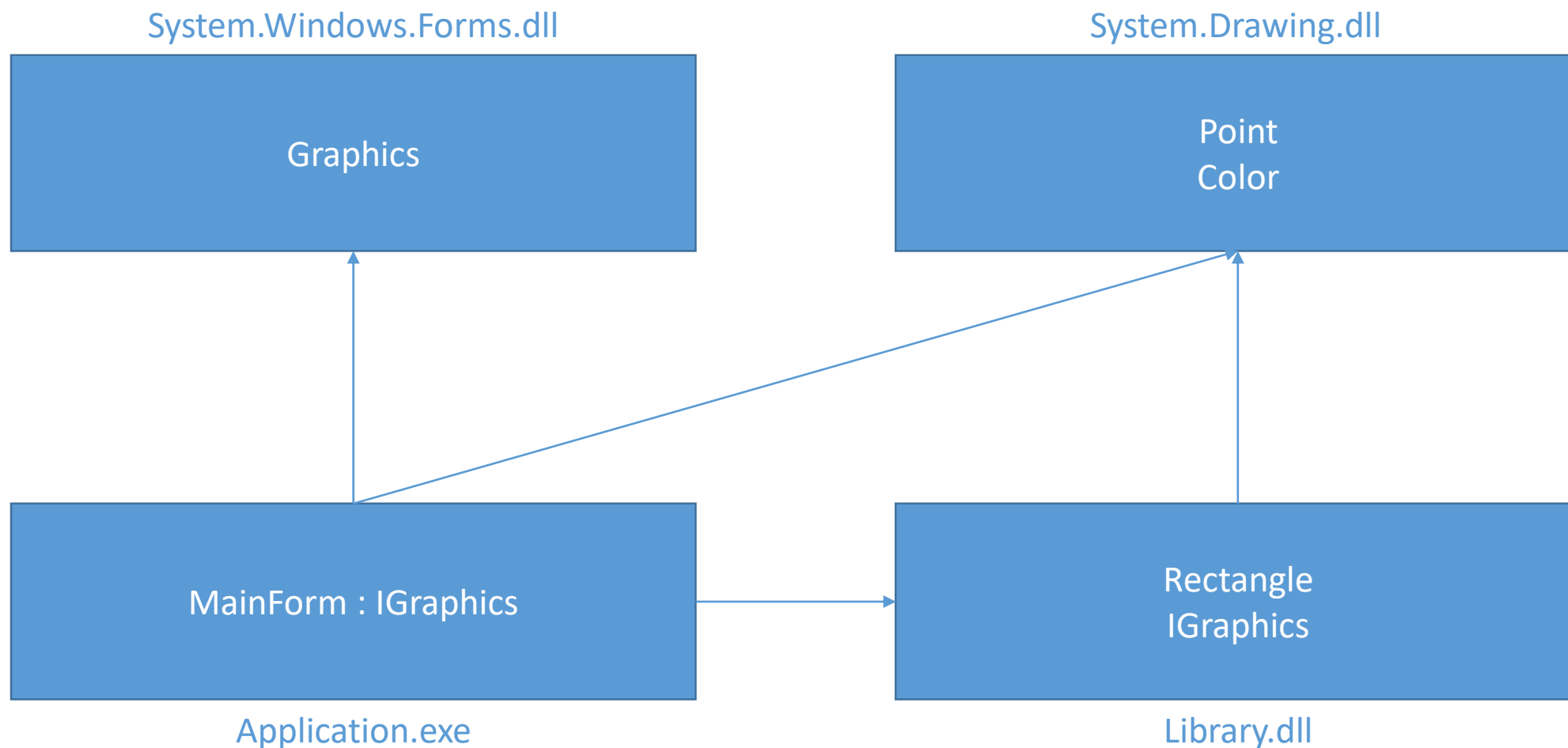
- Ако се премахне зависимостта между Rectangle и Windows.Forms.dll този клас може да се ползва и в проекти с различен потребителски интерфейс (пр: Web интерфейс)

Зависимости между библиотеками



Как да се промени метода Paint, за да бъде
независим от типа Graphics
(Windows.Forms.dll)?

Как да се промени метода Paint, за да бъде независим от типа Graphics?



IGraphics

```
public partial class FormMain : Form, IGraphics
{
    public void DrawRectangle(
        Color color,
        int x, int y,
        int width, int height)
    {
        ...
    }
    ...
    protected override void OnPaint(PaintEventArgs e)
    {
        base.OnPaint(e);

        foreach (var rectangle in _rectangles
            .OrderBy(o => o.Order))
        {
            rectangle.Paint(this);
        }
    }
}
```

Application.exe

```
public interface IGraphics
{
    void DrawRectangle(
        Color color,
        int x, int y,
        int Width, int Height);
}
```

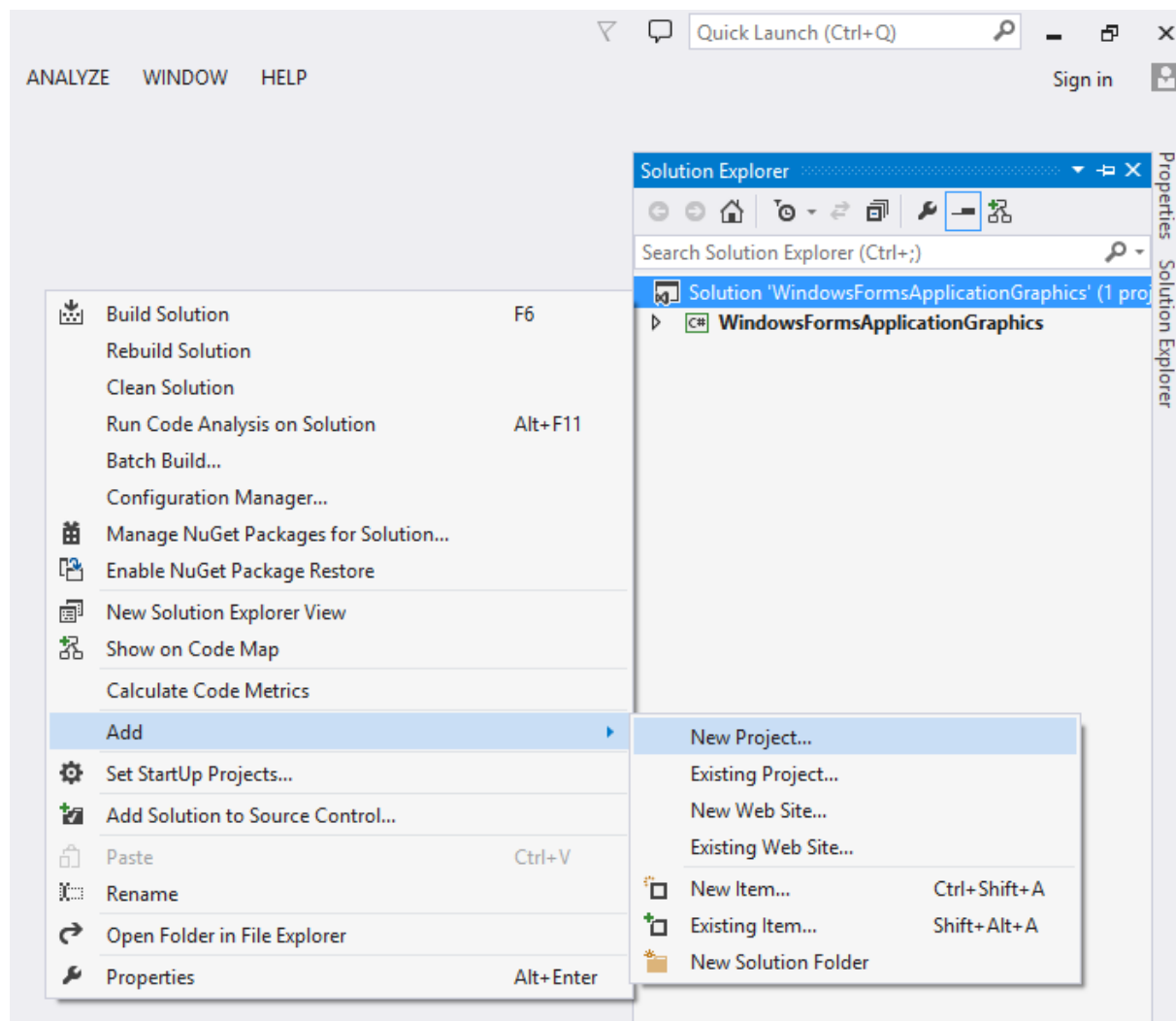
```
public class Rectangle
{
    public void Paint(IGraphics graphics)
    {
        graphics.DrawRectangle(
            Color,
            Position.X, Position.Y,
            Width, Height);
    }
}
```

Library.dll

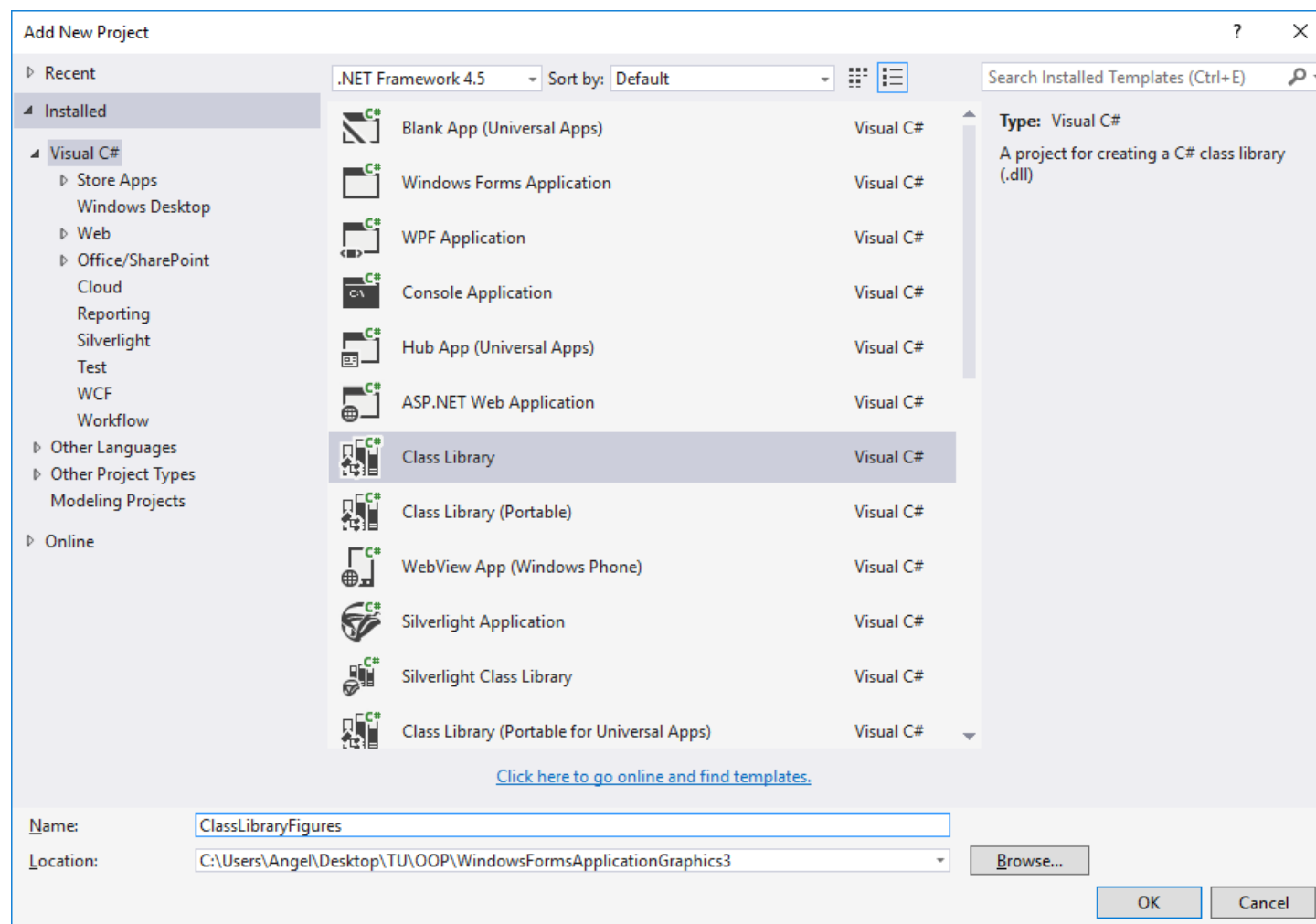
Да се модифицира програмата

- Да се добави проект за библиотека
- Да се добави референция в библиотеката към асемблито System.Drawing.dll
- Да се добави референция в проекта на програмата към проекта на библиотеката
- Да се добави интерфейс IGraphics с метод DrawRectangle приемащ цвят, координати и размери на четириъгълник
- Да се модифицира метода Paint на четириъгълника, да приема променлива от тип IGraphics и да използва метода DrawRectangle
- Да се имплементира IGraphics в класа на формата

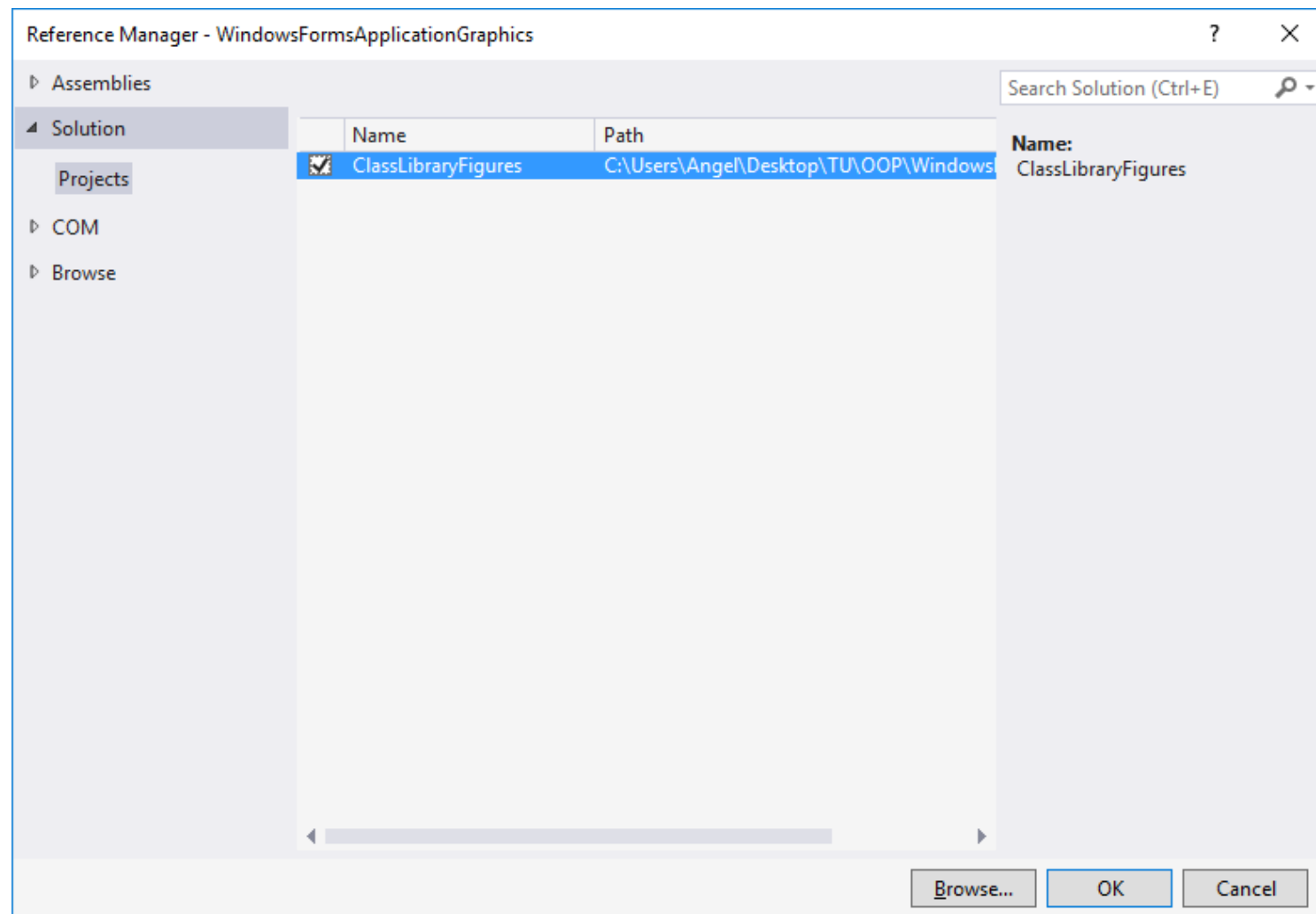
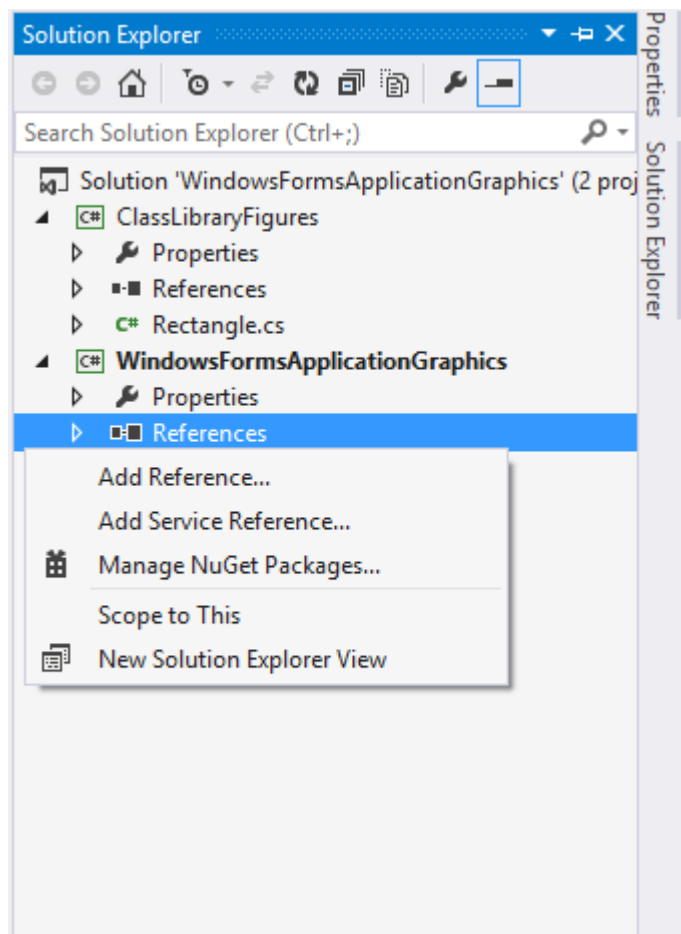
Добавяне на проект за библиотека към проект



Добавяне на проект за библиотека към проект



Добавяне на референция към асембли



Да се модифицира програмата

```
public void DrawRectangle(Color color, int x, int y, int width, int height)
{
    using (var graphics = CreateGraphics())
    using (var pen = new Pen(color))
    {
        graphics.DrawRectangle(pen, x, y, width, height);
    }
}
```