**№ 12 Рефлексия**

**Курносенко Софья**

**Задание**

1. Для изучения .NET Reflection API напишите статический класс ***Reflector***, который собирает информацию и будет содержать методы выполняющие исследования класса (принимают в качестве параметра имя класса) и записывающие информацию в файл (формат тестовый, json или xml):

a. Определение имени сборки, в которой определен класс;

b. есть ли публичные конструкторы;

c. извлекает все общедоступные публичные методы класса (возвращает *IEnumerable*<*string*>);

d. получает информацию о полях и свойствах класса (возвращает *IEnumerable*<*string*>);

e. получает все реализованные классом интерфейсы (возвращает *IEnumerable*<*string*>);

f. выводит по имени класса имена методов, которые содержат заданный (пользователем) тип параметра (имя класса передается в качестве аргумента);

g. метод ***Invoke***, который вызывает метод класса, при этом значения для его параметров необходимо: 1) прочитать из текстового файла (имя класса и имя метода передаются в качестве аргументов) 2) сгенерировать, используя генератор значений для каждого типа. Параметрами метода ***Invoke*** должны быть: объект, имя метода, массив параметров.

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection

Продемонстрируйте работу «Рефлектора» для исследования типов на созданных вами классах не менее двух (предыдущие лабораторные работы) и стандартных классах .Net.

2. Добавьте в ***Reflector*** обобщенный метод ***Create***, который создает объект переданного типа (на основе имеющихся публичных конструкторов) и возвращает его пользователю.

**► Статические классы в C#**

Класс считается статическим, если при его создании, перед ключевым словом **class** указывается ключевое слово **static**. Определение класса статическим, означает то, что этот класс является набором только статических сущностей (полей, методов, свойств и т.п.), и, **внимание**, создавать объекты такого класс нельзя!

Кроме этого, на статический класс, накладываются как минимум следующие ограничения:

* статический класс не может служить базовым для других классов, а так же, сам не может быть наследником какого-либо произвольного класса;
* статический класс не может содержать сущности с атрибутами доступа **protected** и protected **internal**(да они и не нужны, все равно наследование запрещено);
* класс не может иметь перегруженных операций.

Рассмотрим класс «*Operations»*:

//Класс, реализующий некие операции

class Operations

{

//Статический метод, принимающий два целых числа и возвращающий их сумму

public static int Sum(int aFirstArg, int aSecondArg)

{

return aFirstArg + aSecondArg;

}

}

Этот класс не имеет никаких не статических методов, полей свойств и прочего, и он отлично подходит для того, чтобы сделать его статическим. Ведь действительно, создавать объекты этого класса практически бессмысленно! Вот так будет выглядеть этот класс, если мы сделаем его статическим:

//Класс, реализующий некие операции

static class Operations

{

//Статический метод, принимающий два целых числа и возвращающий их сумму

public static int Sum(int aFirstArg, int aSecondArg)

{

return aFirstArg + aSecondArg;

}

}

*Обратите внимание, даже не смотря на то, что весь класс является статическим, метод «Sum» мы тоже явно сделали статическим! Так требует компилятор…*

# ► Рефлексия

## Введение в рефлексию. Класс System.Type

**Рефлексия** представляет собой процесс выявления типов во время выполнения приложения. Каждое приложение содержит набор используемых классов, интерфейсов, а также их методов, свойств и прочих кирпичиков, из которых складывается приложение. И рефлексия как раз и позволяет определить все эти составные элементы приложения.

Основной функционал рефлексии сосредоточен в пространстве имен **System.Reflection**. В нем мы можем выделить следующие основные классы:

* **Assembly**: класс, представляющий сборку и позволяющий манипулировать этой сборкой

**Основные элементы класса Assembly**

****

* **AssemblyName**: класс, хранящий информацию о сборке
* **MemberInfo**: базовый абстрактный класс, определяющий общий функционал для классов EventInfo, FieldInfo, MethodInfo и PropertyInfo
* **EventInfo**: класс, хранящий информацию о событии
* **FieldInfo**: хранит информацию об определенном поле типа
* **MethodInfo**: хранит информацию об определенном методе
* **PropertyInfo**: хранит информацию о свойстве
* **ConstructorInfo**: класс, представляющий конструктор
* **Module**: класс, позволяющий получить доступ к определенному модулю внутри сборки
* **ParameterInfo**: класс, хранящий информацию о параметре метода

Эти классы представляют составные блоки типа и приложения: методы, свойства и т.д. Но чтобы получить информацию о членах типа, нам надо воспользоваться классом **System.Type**.

Класс System.Type представляет изучаемый тип, инкапсулируя всю информацию о нем. С помощью его свойств и методов можно получить эту информацию. Некоторые из его свойств и методов:

* Метод **FindMembers()** возвращает массив объектов MemberInfo данного типа
* Метод **GetConstructors()** возвращает все конструкторы данного типа в виде набора объектов ConstructorInfo
* Метод **GetEvents()** возвращает все события данного типа в виде массива объектов EventInfo
* Метод **GetFields()** возвращает все поля данного типа в виде массива объектов FieldInfo
* Метод **GetInterfaces()** получает все реализуемые данным типом интерфейсы в виде массива объектов Type
* Метод **GetMembers()** возвращает все члены типа в виде массива объектов MemberInfo
* Метод **GetMethods()** получает все методы типа в виде массива объектов MethodInfo
* Метод **GetProperties()** получает все свойства в виде массива объектов PropertyInfo
* Свойство **Name** возвращает имя типа
* Свойство **Assembly** возвращает название сборки, где определен тип
* Свойство **Namespace** возвращает название пространства имен, где определен тип
* Свойство **IsArray** возвращает true, если тип является массивом
* Свойство **IsClass** возвращает true, если тип представляет класс
* Свойство **IsEnum** возвращает true, если тип является перечислением
* Свойство **IsInterface** возвращает true, если тип представляет интерфейс

### Получение типа

Чтобы управлять типом и получать всю информацию о нем, надо сперва получить данный тип. Это можно сделать тремя способами: с помощью ключевого слова **typeof**, с помощью метода **GetType()** класса Object и применяя статический метод Type.GetType().

Получение типа через typeof:

namespace Laba12

{

class Program

{

static void Main(string[] args)

{

Type myType = typeof(User);

Console.WriteLine(myType.ToString());

Console.ReadLine();

}

}

public class User

{

public string Name { get; set; }

public int Age { get; set; }

public User(string n, int a)

{

Name = n;

Age = a;

}

public void Display()

{

Console.WriteLine($"Имя: {Name}  Возраст: {Age}");

}

public int Payment(int hours, int perhour)

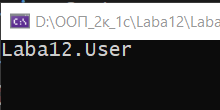
{

return hours \* perhour;

}

}

}



Здесь определен класс User с некоторым набором свойств и полей. И чтобы получить его тип, используется выражение Type myType = typeof(User);

Получение типа с помощью метода GetType класса Object:

User user = new User("Tom", 30);

Type myType = user.GetType();

В отличие от предыдущего примера здесь, чтобы получить тип, надо создавать объект класса.

И третий способ получения типа - статический метод Type.GetType():

Type myType = Type.GetType("Laba12.User", false, true);

Первый параметр указывает на полное имя класса с пространством имен. В данном случае класс User находится в пространстве имен Laba12. Второй параметр указывает, будет ли генерироваться исключение, если класс не удастся найти. В данном случае значение false означает, что исключение не будет генерироваться. И третий параметр указывает, надо ли игнорировать регистр символов в первом параметре. Значение true означает, что регистр игнорируется.

В данном случае класс основной программы и класс User находятся в одном проекте и компилируются в одну сборку exe. Однако может быть, что нужный нам класс находится в другой сборке dll. Для этого после полного имени класса через запятую указывается имя сборки:

|  |
| --- |
| Type myType = Type.GetType("TestConsole.User, MyLibrary", false, true); |

Метод System.Type.GetType() предназначен для получения экземпляра типа, который задан параметром. Параметр метода есть тип string. Параметр указывает название типа, для которого нужно получить информацию. В общем случае, метод System.Type.GetType() имеет 7 перегруженных реализаций.

Чтобы получить перечень методов некоторого типа (например, класса) нужно сначала получить объект, содержащий информацию о типе. Это делается с помощью метода

GetType("NameSpace.TypeName")

где

* *NameSpace* – название пространства имен, в котором размещается тип с именем *TypeName*, требующий исследования;
* *TypeName* – название типа, о котором нужно получить информацию. Типом может быть класс, структура, интерфейс, перечисление, делегат.

##### **Что такое рефлексия методов? Как получить информацию о методах? Пример рефлексии методов**

Рефлексия методов позволяет получить информацию о перечне общедоступных (public) методов из заданного типа. Информацию о методах можно получать для класса, структуры или интерфейса. Только эти типы могут содержать объявления методов.  
Чтобы получить перечень методов заданного экземпляра типа, нужно вызвать метод GetMethods(), возвращающий массив типа MethodInfo, который содержит всю необходимую информацию о методах.

**Пример**. Задан класс Date, который описывает дату. В классе реализованы 3 внутренние переменные и 6 методов доступа к ним. Ниже приведен текст программы, которая демонстрирует рефлексию методов.

using System;

// подключить пространство имен System.Reflection

using System.Reflection;

namespace ConsoleApp2

{

// класс, который описывает дату

public class Date

{

int number;

int month;

int year;

//методы

public int GetNumber() { return number; }

public int GetMonth() { return month; }

public int GetYear() { return year; }

public void SetNumber(int nnumber) { number = nnumber; }

public void SetMonth(int nmonth) { month = nmonth; }

public void SetYear(int nyear) { year = nyear; }

}

class Program

{

static void Main(string[] args)

{

// получить экземпляр типа по его имени

Type tp = Type.GetType("ConsoleApp2.Date"); // название класса - "Date" в сборке ConsoleApp2

// получить массив методов класса Date

MethodInfo[] methods = tp.GetMethods();

// вывести названия методов на экран

int i = 0;

foreach (MethodInfo mi in methods)

{

i++;

Console.WriteLine($"Method[{i}] = {mi.Name}");

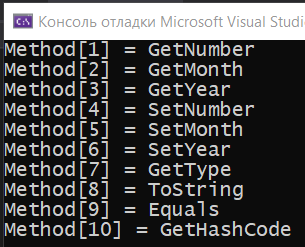
}

}

}

}

В результате выполнения данного кода, на экран будет выведен перечень названий методов, которые доступны для любого объекта класса Date:



Следует отметить следующее:

* информацию можно получить только о тех методах, которые объявленны как общедоступные с модификатором доступа public;
* чтобы получить экземпляр типа методом GetType() из статической функции Main(), обязательно нужно перед именем класса указывать название пространства имен в котором этот класс реализован.

##### **Как получить информацию о полях класса, структуры или перечисления? Примеры рефлексии полей и свойств**

Чтобы получить информацию о поле (свойстве) некоторого типа (класс, структура, перечисление), нужно использовать метод Type.GetFields(). Этот метод возвращает массив типа FieldInfo. Тип FieldInfo содержит всю необходимую информацию о полях и свойствах, которые объявлены как public.

**Пример 1**. Получение информации о полях структуры. Задана структура Worker. В структуре реализованы 3 поля с именами name, age, rank. В примере демонстрируется вывод информации о названии поля и его типе (свойство FieldType).

using System;

// подключить System.Reflection

using System.Reflection;

namespace TrainReflection

{

// структура, которая реализует информацию о работнике

public struct Worker

{

public static string name; // имя работника

public int age; // век работника

public float rank; // рейтинг работника

}

class Program

{

static void Main(string[] args)

{

// получить информацию о полях структуры Worker

// создать объект, который содержит информацию о структуре Worker

Type tp = Type.GetType("TrainReflection.Worker");

// получить информацию о полях структуры Worker

FieldInfo[] fields = tp.GetFields();

// вывести названия полей, их типы и некоторые атрибуты

int i = 0;

string nameType;

foreach (var field in fields)

{

i++;

nameType = field.FieldType.ToString(); // получить название типа поля field структуры Worker

Console.Write($"Field[{i}] = {field.Name}, type = {nameType}");

// проверка, статическое ли это поле

if (field.IsStatic) Console.Write(", static");

// проверка, объявлено ли поле как public

if (field.IsPublic) Console.Write(", public");

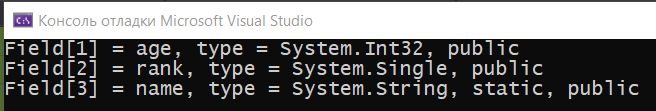
Console.WriteLine();

}

}

}

}



Таким же образом можно получить информацию о полях любого класса любой сборки. Достаточно знать только имя класса в сборке.

**Пример 2**. Получение информации о полях перечисления. Задано перечисление Month, реализующее сокращенные названия месяцев года. Ниже приведен пример получения информации о перечислении на основе информации о его имени.

using System;

// подключить System.Reflection

using System.Reflection;

namespace TrainReflection

{

// перечисление, реализующее информацию о месяце года

enum Month

{

Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec

}

class Program

{

static void Main(string[] args)

{

// получить информацию о полях перечисления Month

// создать объект, который содержит информацию о структуре Worker

Type tp = Type.GetType("TrainReflection.Month");

// получить информацию о полях перечисления Month

FieldInfo[] fields = tp.GetFields();

// вывести названия полей перечисления enum

int i = 0;

foreach (var field in fields)

{

i++;

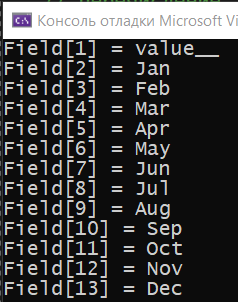
Console.WriteLine($"Field[{i}] = {field.Name}");

}

}

}

}



**Как получить информацию об интерфейсах, которые реализует класс? Пример рефлексии интерфейсов**

Чтобы получить перечень имен интерфейсов, которые поддерживаются заданным типом, используется метод GetInterfaces(). Этот метод возвращает массив интерфейсов типа Type[].  
Название интерфейса помещается в свойстве Name.

**Пример.** Пусть задан интерфейс IDate, в котором объявляются имена двух методов: GetNumber(), SetNumber(). Также объявляется класс Date, реализующий дату. Класс Date наследует интерфейс IDate. В примере продемонстрировано:

* получение списка интерфейсов класса Date;
* получение списка методов интерфейса IDate.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

// подключить System.Reflection

using System.Reflection;

namespace TrainReflection

{

// интерфейс с двумя объявленными методами

interface IDate

{

int GetNumber();

void SetNumber(int nnumber);

}

// класс, который реализует интерфейс IDate

public class Date : IDate

{

int number;

int month;

int year;

//

public int GetNumber() { return number; }

public int GetMonth() { return month; }

public int GetYear() { return year; }

public void SetNumber(int nnumber) { number = nnumber; }

public void SetMonth(int nmonth) { month = nmonth; }

public void SetYear(int nyear) { year = nyear; }

}

class Program

{

static void Main(string[] args)

{

// получить информацию о методах

// получить экземпляр типа по его имени

Type tp = Type.GetType("TrainReflection.Date"); // название класса - "Date"

// получить массив интерфейсов класса Date

Type[] interfaces = tp.GetInterfaces();

// вывести названия интерфейсов класса Date на экран

int i = 0;

foreach (Type t in interfaces)

{

i++;

Console.WriteLine("Interface[{0}] = {1}", i, t.Name);

}

Console.WriteLine();

// создать объект, который содержит информацию об интерфейсе IDate

Type tpi = Type.GetType("TrainReflection.IDate");

// получить массив методов для интерфейса IDate с помощью объекта tpi

MethodInfo[] methods = tpi.GetMethods();

// вывести перечень методов интерфейса IDate

i = 0;

foreach (MethodInfo mt in methods)

{

i++;

// вывести полное имя каждого метода интерфейса

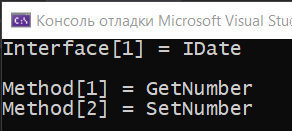
Console.WriteLine("Method[{0}] = {1}", i, mt.Name);

}

}

}

}



##### **Пример использования рефлексии для получения дополнительных сведений о типе (классе, структуре, перечислении)**

С помощью рефлексии можно получать дополнительные сведения о типе. Типом может быть класс, структура, интерфейс, перечисление, делегат.

**Пример**. Пусть заданы два класса:

* класс Point, который реализует точку на координатной плоскости;
* класс Pixel, который реализует точку заданного цвета. Класс Pixel унаследован от класса Point.

Ниже приводится полный текст программы, которая выводит дополнительные сведения о классах Point и Pixel.

using System;

namespace TrainReflection

{

public class Point

{

// внутренние переменные класса

protected int x;

protected int y;

// методы доступа

public void GetXY(out int xx, out int yy)

{

xx = x;

yy = y;

}

public void SetXY(int xx, int yy)

{

x = xx;

y = yy;

}

}

// класс Pixel является производным от класса Point

sealed class Pixel : Point

{

// внутренняя переменная color

int color;

// методы доступа

public void GetXYC(out int xx, out int yy, out int cl)

{

GetXY(out xx, out yy);

cl = color;

}

public void SetXYC(int xx, int yy, int cl)

{

SetXY(xx, yy);

color = cl;

}

}

class Program

{

static void Main(string[] args)

{

// создать объекты для классов Point и Pixel

Type tPoint = Type.GetType("TrainReflection.Point");

Type tPixel = Type.GetType("TrainReflection.Pixel");

// вывести некоторую информацию о классе Point

Console.WriteLine("Information about Point class:");

if (tPoint.IsAbstract) Console.WriteLine("Abstract class"); // абстрактный класс

if (tPoint.IsArray) Console.WriteLine("Array"); // это есть массив

if (tPoint.IsClass) Console.WriteLine("Class or delegate"); // это есть класс или делегат

if (tPoint.IsEnum) Console.WriteLine("Enumeration"); // это есть перечисление

if (tPoint.IsInterface) Console.WriteLine("Interface"); // это есть интерфейс

if (tPoint.IsPublic) Console.WriteLine("Public"); // объявлен как public

if (tPoint.IsSealed) Console.WriteLine("Sealed"); // запечатан

Console.WriteLine();

// вывести некоторую информацию о классе Pixel

Console.WriteLine("Information about Pixel class:");

if (tPixel.IsAbstract) Console.WriteLine("Abstract class"); // абстрактный класс

if (tPixel.IsArray) Console.WriteLine("Array"); // это есть массив

if (tPixel.IsClass) Console.WriteLine("Class or delegate"); // это есть класс или делегат

if (tPixel.IsEnum) Console.WriteLine("Enumeration"); // это есть перечисление

if (tPixel.IsInterface) Console.WriteLine("Interface"); // это есть интерфейс

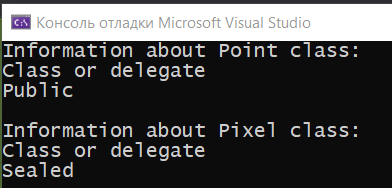
if (tPixel.IsPublic) Console.WriteLine("Public"); // объявлен как public

if (tPixel.IsSealed) Console.WriteLine("Sealed"); // запечатан

}

}

}



**Как получить информацию о свойствах (properties) класса? Пример**

Чтобы получить информацию о свойствах класса нужно вызвать метод GetProperties(). Этот метод возвращает массив типа PropertyInfo[]. Массив PropertyInfo[] содержит сведения о свойствах (properties) класса.

**Пример**. Продемонстрировано получение перечня свойств класса Student из пространства имен TrainReflection. Свойства обязательно должны быть объявлены с модификатором доступа public.

using System;

// подключить System.Reflection

using System.Reflection;

namespace TrainReflection

{

// информация о студенте

class Student

{

string name; // имя студента

string numBook; // номер зачетной книжки

float rank; // рейтинг студента

// свойства

public string Name

{

get { return name; }

set { name = value; }

}

public string NumBook

{

get { return numBook; }

set { numBook = value; }

}

public float Rank

{

get { return rank; }

set { rank = value; }

}

}

class Program

{

static void Main(string[] args)

{

// получить информацию о свойствах из класса Student

// получить объект типа System.Type

Type tp = Type.GetType("TrainReflection.Student");

// получить перечень свойств

PropertyInfo[] properties = tp.GetProperties();

// вывести перечень свойств

int i = 0;

foreach (PropertyInfo p in properties)

{

i++;

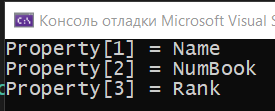
Console.WriteLine("Property[{0}] = {1}", i, p.Name);

}

}

}

}



**Пример использования рефлексии для получения информации о параметрах метода**

Для конкретного метода можно получить информацию о его параметрах.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

// подключить System.Reflection

using System.Reflection;

namespace TrainReflection

{

// класс, который содержит математические функции

class MathFunctions

{

// метод, который решает квадратное уравнение

public bool CalcEquation(double a, double b, double c, out double x1, out double x2)

{

double d = b \* b - 4 \* a \* c;

if (a == 0)

{

if (b == 0)

{

x1 = x2 = 0;

return false;

}

else

{

x1 = x2 = -c / b;

return true;

}

}

if (d < 0)

{

x1 = x2 = 0;

return false;

}

x1 = (-b - Math.Sqrt(d)) / (2 \* a);

x2 = (-b + Math.Sqrt(d)) / (2 \* a);

return true;

}

}

class Program

{

static void Main(string[] args)

{

// получить информацию о свойствах из класса Student

// получить объект типа System.Type

Type tp = Type.GetType("TrainReflection.MathFunctions");

// получить объект, который содержит информацию о конкретном методе CalcEquation класса MathFunctions

MethodInfo method = tp.GetMethod("CalcEquation");

// получить параметры метода CalcEquation

ParameterInfo[] parameters = method.GetParameters();

// вывести параметры метода

Console.WriteLine("Parameters of method CalcEquation:");

int i = 0;

foreach (ParameterInfo p in parameters)

{

i++;

Console.WriteLine("Parameter[{0}] = {1} of type {2}", i, p.Name, p.ParameterType.Name);

}

// вывести возвращаемый методом тип

string returnType = method.ReturnType.FullName;

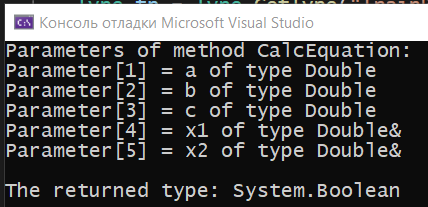
Console.WriteLine();

Console.WriteLine("The returned type: {0}", returnType);

}

}

}



**Получение имени сборки, где определен класс**

# Assembly.GetAssembly получает сборку, в которой определен указанный тип.

Type typeInfo = Type.GetType(type);

string assemblyName = Assembly.GetAssembly(typeInfo).FullName;

### Получение конструкторов

Для получения конструкторов применяется метод ConstructorInfo[] GetConstructors():

Type myType = Type.GetType("HelloApp.User", false, true);

Console.WriteLine("Конструкторы:");

foreach (ConstructorInfo ctor in myType.GetConstructors())

{

Console.Write(myType.Name + " (");

// получаем параметры конструктора

ParameterInfo[] parameters = ctor.GetParameters();

for (int i = 0; i < parameters.Length; i++)

{

Console.Write(parameters[i].ParameterType.Name + " " + parameters[i].Name);

if (i + 1 < parameters.Length) Console.Write(", ");

}

Console.WriteLine(")");

}

**Получение определенных членов типа**

# BindingFlags Enum

Задает флаги, которые управляют привязкой и способом, которым выполняется поиск членов и типов путем отражения.

Перечисление **BindingFlags** может принимать различные значения:

* **DeclaredOnly**: получает только члены непосредственно данного класса, унаследованные члены не извлекаются
* **Instance**: получает только члены экземпляра
* **NonPublic**: извлекает не публичные члены
* **Public**: получает только публичные члены
* **Static**: получает только статические члены

Объединяя данные значения с помощью побитовой операции ИЛИ или И можно комбинировать вывод. Например, получим только публичные или не унаследованные методы класса:

MethodInfo[] methods = myType.GetMethods(BindingFlags.DeclaredOnly | BindingFlags.Public);

**► Позднее и раннее связывание**

**(или статическая и динамическая типизация)**

Если классы образовывают иерархию наследования, то при обращении к элементам класса, компилятор может реализовывать один из двух возможных способов связывания кода:

1. *Раннее связывание*– связанное с формированием кода на этапе компиляции. При раннем связывании, программный код формируется на основе известной информации о типе (класс) ссылки. Как правило, это ссылка на базовый класс в иерархии классов.
2. *Позднее связывание*– связанное с формированием кода на этапе выполнения. Если в иерархии классов встречается цепочка виртуальных методов (с помощью слов virtual, override), то компилятор строит так называемое позднее связывание. При позднем связывании вызов метода происходит на основании типа объекта, а не типа ссылки на базовый класс. Позднее связывание используется, если нужно реализовать *полиморфизм*.

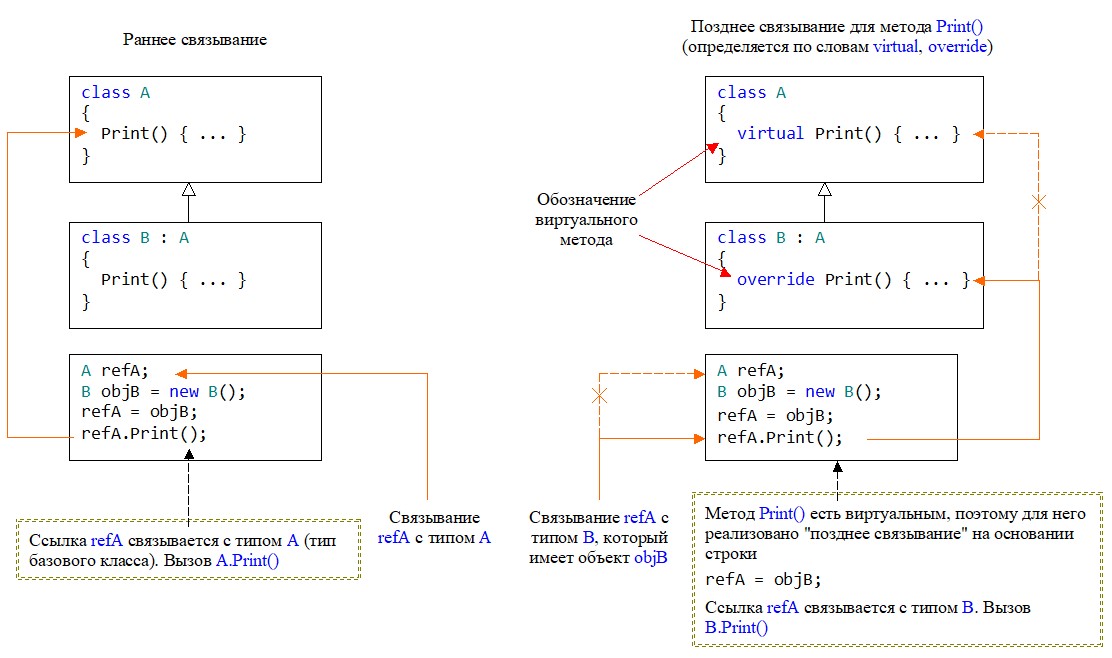
Выбор того или иного вида связывания для каждого отдельного элемента (метода, свойства, индексатора и т.п.) определяется компилятором по следующим правилам:

* если в иерархии унаследованных классов объявляется невиртуальный элемент, то реализуется раннее связывание;
* если в иерархии унаследованных классов объявляется виртуальный элемент, то выполняется позднее связывание. Виртуальный элемент в базовом классе обозначается ключевым словом virtual, во всех унаследованных классах ключевым словом override. В C# виртуальным элементом может быть метод, событие, индексатор или свойство.

**Необходимые условия для реализации позднего связывания:**

* классы должны образовывать иерархию наследования;
* в классах должны быть методы с одинаковой сигнатурой. Элементы (методы) производных классов должны перекрывать (override) соответствующие элементы (методы) базовых классов;
* элементы (методы) класса должны быть виртуальными, то есть должны быть обозначены ключевыми словами virtual, override.

На рисунке приведен пример, который отображает отличие между поздним и ранним связыванием на примере двух классов A, B в которых реализован метод Print().



В случае раннего связывания, как только компилятор встречает строку

A ref;

происходит объявление ссылки ref, которая имеет тип базового класса A. Дальнейшее присваивание

refA = objB;

связывает ссылку с объектом objB, однако тип ссылки устанавливается A. Поэтому вызов

ref.Print();

вызовет метод Print() класса A.

В случае позднего связывания, сначала на основе описания классов A, B компилятор определяет, что метод Print() является виртуальным. Для виртуального метода компилятор строит таблицу виртуальных методов Print(), которая содержит смещение адресов каждого виртуального метода для каждого класса иерархии.

После строки

A ref;

формируется связывание ссылки ref с типом A. После присваивания

ref = objB;

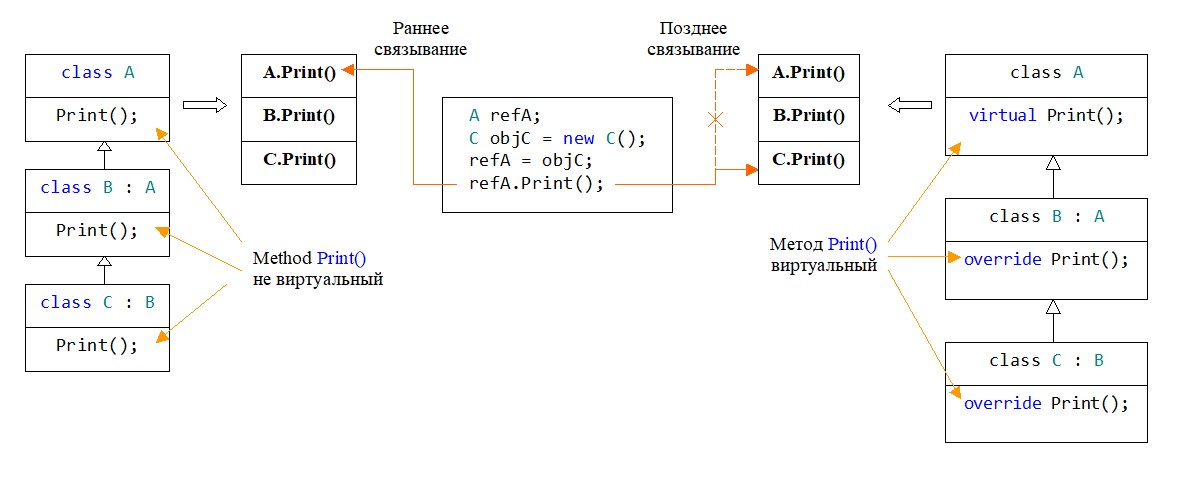
компилятор присваивает ссылке ref адрес экземпляра objB и определяет тип связывания как тип B (поскольку метод Print() виртуальный). За основу берется тип объекта. В результате ссылка ref связывается с методом Print(), который реализован в классе B (а не в классе A) – выполняется так называемое «позднее связывание».

Как следствие, после вызова

ref.Print();

будет вызван метод Print() класса B.

На рисунке приведено отличие между поздним и ранним связываниями на примере трех классов A, B, C, в каждом из которых объявлен метод Print().



При раннем связывании компилятор находит нужный метод, а то, какой нужен метод, определяется типом ссылки. Затем вызывается именно метод класса ссылки.

using System;

namespace pract

{

class A

{

public void Print()

{

Console.WriteLine("I'm \"A\" class");

}

}

class B : A

{

// сокрытие метода Print базового класса ()A в классе наследнике (B)

public new void Print()

{

Console.WriteLine("I'm \"B\" class");

}

}

class Program

{

static void Main(string[] args)

{

// ссылке базового класса присваиваем объект класса наследника

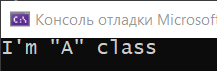
A obj = new B();

obj.Print();

}

}

}



При позднем связывании мы говорим, что компилятор при выполнении программы должен найти метод с определенным именем и вызвать его. А то, что это будет за метод, определяется типом объекта.

using System;

namespace pract

{

class A

{

public virtual void Print()

{

Console.WriteLine("I'm \"A\" class");

}

}

class B : A

{

// сокрытие метода Print базового класса ()A в классе наследнике (B)

public override void Print()

{

Console.WriteLine("I'm \"B\" class");

}

}

class Program

{

static void Main(string[] args)

{

// ссылке базового класса присваиваем объект класса наследника

A obj = new B();

obj.Print();

}

}

}

