Теория и практика программирования

Шпилёв Пётр Валерьевич

Санкт-Петербургский государственный университет
Математико-механический факультет
Кафедра статистического моделирования

Лекция 9

Санкт-Петербург
2015 г.

e-mail: pitshp@hotmail.com

Общая форма объявления пространства имен

```
namespace имя
{
  //члены
}
```

Общая форма объявления пространства имен

```
namespace имя
{
  //члены
}
```

Замечание

Доступ к членам пространства имен осуществляется следующим образом: имя_пространства_имен.имя_члена

Общая форма объявления пространства имен

```
namespace имя
{
  //члены
}
```

Пример:

```
namespace NStest
{
    class A {... }
    class B {... }
    class C {... }
}
class NSDemo
{
    static void Main()
    {
        NStest.A a1 = new NStest.A();
        NStest.B b1 =new NStest.B();
        NStest.C c1 = new NStest.C();
    }
}
```

Общая форма объявления пространства имен

```
namespace имя
{
  //члены
}
```

Упражнение 9.1

Добавить в одну из своих программы несколько пространств имен и реализовать обращения к их членам. В качестве членов разных пространств имен использовать классы с одинаковыми именами.

Директива using

С помощью директивы using можно сделать видимыми вновь создаваемые пространства имен. Существуют две формы этой директивы:
using имя;
using псевдоним = имя;

Директива using

С помощью директивы using можно сделать видимыми вновь создаваемые пространства имен. Существуют две формы этой директивы:
using имя;
using псевдоним = имя;

Пример:

```
using ClassAFromNStest = NStest.A;
namespace NStest
{
    class A {... }
}
class NSDemo
{
    ClassAFromNStest aNStest = new ClassAFromNStest();
    ...
}
```

Директива using

С помощью директивы using можно сделать видимыми вновь создаваемые пространства имен. Существуют две формы этой директивы:
using имя;
using псевдоним = имя;

Пример:

```
using ClassAFromNStest = NStest.A;
namespace NStest
{
    class A {... }
}
class NSDemo
{
    ClassAFromNStest aNStest = new ClassAFromNStest();
    ...
}
```

Упражнение 9.2

Для предыдущего упражнения продемонстрировать применение обеих форм директивы using.

Замечание

Под одним именем можно объявить несколько пространств имен. Это дает возможность распределить пространство имен по нескольким файлам или даже разделить его в пределах одного и того же файла исходного кода.

Замечание

Под одним именем можно объявить несколько пространств имен. Это дает возможность распределить пространство имен по нескольким файлам или даже разделить его в пределах одного и того же файла исходного кода.

Две формы задания вложенных пространств имен:

I:
```
namespace OuterNS
{
    namespace InnerNS
    {
        //...
    }
}
```

II:
```
namespace OuterNS.InnerNS
{
    //...
}
```

Замечание

Под одним именем можно объявить несколько пространств имен. Это дает возможность распределить пространство имен по нескольким файлам или даже разделить его в пределах одного и того же файла исходного кода.

Две формы задания вложенных пространств имен:

I:
```
namespace OuterNS
{
    namespace InnerNS
    {
        //...
    }
}
```

II:
```
namespace OuterNS.InnerNS
{
    //...
}
```

Общая форма оператора ::

псевдоним_пространства_имен::идентификатор

Общая форма оператора ::

псевдоним_пространства_имен::идентификатор

Пример:

```
using NSt = NStest;
...
NSt::A cd1 = new NSt::A();
```

Общая форма оператора ::

псевдоним_пространства_имен::идентификатор

Пример:

```
using NSt = NStest;
...
NSt::A cd1 = new NSt::A();
```

Замечание

Описатель :: можно также использовать вместе с предопределенным идентификатором global для ссылки на глобальное пространство имен.

Общая форма оператора ::

псевдоним_пространства_имен::идентификатор

Пример:

```
using NSt = NStest;
...
NSt::A cd1 = new NSt::A();
```

Замечание

Описатель :: можно также использовать вместе с предопределенным идентификатором global для ссылки на глобальное пространство имен.

Упражнение 9.3

Использовать описатель :: для идентификации классов с одинаковыми именами из разных пространств имен

Таблица 1: Директивы препроцессора, определенные в C#.

`#define`	`#elif`	`#else`	`#endif`
`#endregion`	`#error`	`#if`	`#line`
`#pragma`	`#region`	`#undef`	`#warning`

Таблица 1: Директивы препроцессора, определенные в C#.

`#define`	`#elif`	`#else`	`#endif`
`#endregion`	`#error`	`#if`	`#line`
`#pragma`	`#region`	`#undef`	`#warning`

Замечание

Термин директива препроцессора появился в связи с тем, что подобные инструкции по традиции обрабатывались на отдельной стадии компиляции, называемой препроцессором. Обрабатывать директивы на отдельной стадии препроцессора в современных компиляторах уже не нужно, но само ее название закрепилось.

Общая форма директивы #define.

#define идентификатор

Общая форма директивы #define.

```
#define идентификатор
```

Общая форма директив #if и #endif

```
#if идентификаторное_выражение
  последовательность операторов
#endif
```

Общая форма директивы #define.

#define идентификатор

Общая форма директив #if и #endif

#if идентификаторное_выражение
  последовательность операторов
#endif

Замечание

Идентификаторное выражение может быть простым, как наименование идентификатора. В то же время в нем разрешается применение следующих операторов: $!$, $==$, $!=$, $\&\&$ *и* $||$, *а также круглых скобок.*

Общая форма директивы #define.

```
#define идентификатор
```

Общая форма директив #if и #endif

```
#if идентификаторное_выражение
  последовательность операторов
#endif
```

Замечание

Идентификаторное выражение может быть простым, как наименование идентификатора. В то же время в нем разрешается применение следующих операторов: !, ==, !=, && *и* ||, *а также круглых скобок.*

Упражнение 9.4

Продемонстрировать применение директив #if, #endif и #define. Использовать идентификаторное выражение.

Общая форма директивы #define.

#define идентификатор

Общая форма директив #if и #endif

#if идентификаторное_выражение
  последовательность операторов
#endif

Замечание

Идентификаторное выражение может быть простым, как наименование идентификатора. В то же время в нем разрешается применение следующих операторов: !, ==, != , && *и* ||, *а также круглых скобок.*

Упражнение 9.4

Продемонстрировать применение директив #if, #endif и #define. Использовать идентификаторное выражение.

Упражнение 9.5

Продемонстрировать применение директив #else и #elif.

Общая форма директивы #undef.

#undef идентификатор

Общая форма директивы #undef.

#undef идентификатор

Общая форма директивы #error.

#error сообщение_об_ошибке

Общая форма директивы #undef.

#undef идентификатор

Общая форма директивы #error.

#error сообщение_об_ошибке

Общая форма директивы #warning.

#warning предупреждающее_сообщение

Общая форма директивы #undef.

#undef идентификатор

Общая форма директивы #error.

#error сообщение_об_ошибке

Общая форма директивы #warning.

#warning предупреждающее_сообщение

Общая форма директивы #line.

#line номер "имя_файла"

Общая форма директивы #undef.

#undef идентификатор

Общая форма директивы #error.

#error сообщение_об_ошибке

Общая форма директивы #warning.

#warning предупреждающее_сообщение

Общая форма директивы #line.

#line номер "имя_файла"

[1] #line default

[2] #line hidden

Общая форма директивы #undef.

```
#undef идентификатор
```

Общая форма директивы #error.

```
#error сообщение_об_ошибке
```

Общая форма директивы #warning.

```
#warning предупреждающее_сообщение
```

Общая форма директивы #line.

```
#line номер "имя_файла"
```

Общая форма директив #region и #endregion

```
#region текст
// последовательность кода
#endregion текст
```

Общая форма директивы #pragma

```
#pragma опция
```

Общая форма директивы #line.

```
#line номер "имя_файла"
```

Общая форма директив #region и #endregion

```
#region текст
// последовательность кода
#endregion текст
```

Общая форма директивы #pragma

```
#pragma опция
```

[1] #pragma warning disable номер_предупреждения

[2] #pragma warning restore номер_предупреждения

[3] #pragma checksum

Общая форма директивы #line.

```
#line номер "имя_файла"
```

Общая форма директив #region и #endregion

```
#region текст
// последовательность кода
#endregion текст
```

Общая форма оператора is

выражение is тип

где *выражение* обозначает отдельное выражение, описывающее объект, тип которого проверяется. Если выражение имеет совместимый или такой же тип, как и проверяемый тип, то результат этой операции получается истинным, в противном случае – ложным.

Общая форма оператора is

выражение is тип
где *выражение* обозначает отдельное выражение, описывающее объект, тип которого проверяется. Если выражение имеет совместимый или такой же тип, как и проверяемый тип, то результат этой операции получается истинным, в противном случае – ложным.

Общая форма оператора as

выражение as тип
где *выражение* обозначает отдельное выражение, преобразуемое в указанный тип. Если исход такого преобразования оказывается удачным, то возвращается ссылка на тип, а иначе — пустая ссылка.

Общая форма оператора is

выражение is тип
где *выражение* обозначает отдельное выражение, описывающее объект, тип которого проверяется. Если выражение имеет совместимый или такой же тип, как и проверяемый тип, то результат этой операции получается истинным, в противном случае – ложным.

Общая форма оператора as

выражение as тип
где *выражение* обозначает отдельное выражение, преобразуемое в указанный тип. Если исход такого преобразования оказывается удачным, то возвращается ссылка на тип, а иначе — пустая ссылка.

Упражнение 9.6

Продемонстрировать применение оператора is.

Общая форма оператора is

выражение is тип
где *выражение* обозначает отдельное выражение, описывающее объект, тип которого проверяется. Если выражение имеет совместимый или такой же тип, как и проверяемый тип, то результат этой операции получается истинным, в противном случае – ложным.

Общая форма оператора as

выражение as тип
где *выражение* обозначает отдельное выражение, преобразуемое в указанный тип. Если исход такого преобразования оказывается удачным, то возвращается ссылка на тип, а иначе — пустая ссылка.

Упражнение 9.6

Продемонстрировать применение оператора is.

Упражнение 9.7

Продемонстрировать применение оператора as.

Рефлексия

Рефлексия — это средство, позволяющее получать сведения о типе данных. Для применения рефлексии в код программы обычно вводится строка:

```
using System.Reflection;
```

Рефлексия

Рефлексия — это средство, позволяющее получать сведения о типе данных. Для применения рефлексии в код программы обычно вводится строка:
using System.Reflection;

Оператор typeof

Данный оператор извлекает объект класса System.Type для заданного типа. Информация, описывающая тип, инкапсулируется в возвращаемом объекте класса Type.

Рефлексия

Рефлексия — это средство, позволяющее получать сведения о типе данных. Для применения рефлексии в код программы обычно вводится строка:
using System.Reflection;

Оператор typeof

Данный оператор извлекает объект класса System.Type для заданного типа. Информация, описывающая тип, инкапсулируется в возвращаемом объекте класса Type.

Общая форма оператора typeof

typeof(тип);
где тип обозначает получаемый тип.

Рефлексия

Рефлексия — это средство, позволяющее получать сведения о типе данных. Для применения рефлексии в код программы обычно вводится строка:
using System.Reflection;

Оператор typeof

Данный оператор извлекает объект класса System.Type для заданного типа. Информация, описывающая тип, инкапсулируется в возвращаемом объекте класса Type.

Общая форма оператора typeof

typeof(тип);
где тип обозначает получаемый тип.

Пример: Type t = typeof(MyClass);

Рефлексия

Рефлексия — это средство, позволяющее получать сведения о типе данных. Для применения рефлексии в код программы обычно вводится строка:
using System.Reflection;

Оператор typeof

Данный оператор извлекает объект класса System.Type для заданного типа. Информация, описывающая тип, инкапсулируется в возвращаемом объекте класса Type.

Общая форма оператора typeof

typeof(тип);
где тип обозначает получаемый тип.

Пример: Type t = typeof(MyClass);

Упражнение 9.8

Продемонстрировать применение оператора typeof используя три свойства класса Type: FullName, IsClass и IsAbstract.

Таблица 2: Свойства абстрактного класса System.Reflection.MemberInfo

Свойство	Описание
Type DeclaringType	Тип класса или интерфейса, в котором объявляется отражаемый член
MemberTypes MemberType	Тип члена. Это значение обозначает, является ли член полем, методом, свойством, событием или конструктором
int MetadataToken	Значение, связанное к конкретными метаданными
Module Module	Объект типа Module, представляющий модуль (исполняемый файл), в котором находится отражаемый
string Name	Имя типа
Type ReflectedType	Тип отражаемого объекта

Таблица 2: Свойства абстрактного класса System.Reflection.MemberInfo

Свойство	Описание
Type DeclaringType	Тип класса или интерфейса, в котором объявляется отражаемый член
MemberTypes MemberType	Тип члена. Это значение обозначает, является ли член полем, методом, свойством, событием или конструктором
int MetadataToken	Значение, связанное к конкретными метаданными
Module Module	Объект типа Module, представляющий модуль (исполняемый файл), в котором находится отражаемый
string Name	Имя типа
Type ReflectedType	Тип отражаемого объекта

Таблица 3: Методы абстрактного класса System.Reflection.MemberInfo

Метод	Назначение
GetCustomAttributes()	Получает список специальных атрибутов, имеющих отношение к вызывающему объекту
IsDefined()	Устанавливает, определен ли атрибут для вызывающего метода

Таблица 4: Методы класса Type

Метод	Назначение
ConstructorInfo[] GetConstructors()	Получает список конструкторов для заданного типа
EventInfo[] GetEvents()	Получает список событий для заданного типа
FieldInfo[] GetFields()	Получает список полей для заданного типа
Type[] GetGenericArguments()	Получает список аргументов типа, связанных с закрыто сконструированным обобщенным типом, или же список параметров типа, если заданный тип определен как обобщенный. Для открыто сконструированного типа этот список может содержать как аргументы, так и параметры типа.
MemberInfo[] GetMembers()	Получает список членов для заданного типа
MethodInfo[] GetMethods()	Получает список методов для заданного типа
PropertyInfo[] GetProperties()	Получает список свойств для заданного типа

Таблица 5: Свойства класса Type

Свойство	Назначение
Assembly Assembly	Получает сборку для заданного типа
TypeAttributes Attributes	Получает атрибуты для заданного типа
Type BaseType	Получает непосредственный базовый тип для заданного типа
string FullName	Получает полное имя заданного типа
bool IsAbstract	Истинно, если заданный тип является абстрактным
bool IsArray	Истинно, если заданный тип является массивом
bool IsClass	Истинно, если заданный тип является классом
bool IsEnum	Истинно, если заданный тип является перечислением
bool IsGenericParameter	Истинно, если заданный тип является параметром обобщенного типа.
bool IsGenericType	Истинно, если заданный тип является обобщенным.
string Namespace	Получает пространство имен для заданного типа

Метод GetMethods()

MethodInfo[] GetMethods()
Этот метод возвращает массив объектов класса MethodInfo, которые описывают методы, поддерживаемые вызывающим типом.

Метод GetMethods()

MethodInfo[] GetMethods()
Этот метод возвращает массив объектов класса MethodInfo, которые описывают методы, поддерживаемые вызывающим типом.

Замечание

Для получения имени метода служит свойство Name. Возвращаемый тип метода находится в доступном только для чтения свойстве ReturnType, которое является объектом класса Type.

Метод GetMethods()

MethodInfo[] GetMethods()
Этот метод возвращает массив объектов класса MethodInfo, которые описывают методы, поддерживаемые вызывающим типом.

Замечание

Для получения имени метода служит свойство Name. Возвращаемый тип метода находится в доступном только для чтения свойстве ReturnType, которое является объектом класса Type.

Метод GetParameters()

ParameterInfo[] GetParameters()
Метод GetParameters() возвращает список параметров, связанных с анализируемым методом.

Метод GetMethods()

MethodInfo[] GetMethods()
Этот метод возвращает массив объектов класса MethodInfo, которые описывают методы, поддерживаемые вызывающим типом.

Замечание

Для получения имени метода служит свойство Name. Возвращаемый тип метода находится в доступном только для чтения свойстве ReturnType, которое является объектом класса Type.

Метод GetParameters()

ParameterInfo[] GetParameters()
Метод GetParameters() возвращает список параметров, связанных с анализируемым методом.

Замечание

Особое значение имеют два свойства класса ParameterInfo : Name — представляет собой строку, содержащую имя параметра, а ParameterType — описывает тип параметра, который инкапсулирован в объекте класса Type.

Метод GetMethods()

MethodInfo[] GetMethods()
Этот метод возвращает массив объектов класса MethodInfo, которые описывают методы, поддерживаемые вызывающим типом.

Замечание

Для получения имени метода служит свойство Name. Возвращаемый тип метода находится в доступном только для чтения свойстве ReturnType, которое является объектом класса Type.

Метод GetParameters()

ParameterInfo[] GetParameters()
Метод GetParameters() возвращает список параметров, связанных с анализируемым методом.

Упражнение 9.9

Провести анализ методов своего класса с помощью рефлексии (используя методы GetMethods() и GetParameters(), и свойства Name, ReturnType и ParameterType)

Метод Invoke()

```
object Invoke(object obj, object[] parameters)
```

Метод Invoke()

```
object Invoke(object obj, object[] parameters)
```

Упражнение 9.10

Вызвать методы своего класса с помощью рефлексии (используя метод Invoke()).

Пример:

```
class MyClass
{
    public int MyMethod(){...}
...
}
class InvokeMethDemo
{
    static void Main()
    {
        Type t = typeof(MyClass);
        MyClass reflectOb = new MyClass(10, 20);
        MethodInfo[] mi = t.GetMethods();
        foreach(MethodInfo m in mi) //Вызвать метод.
        {
            ParameterInfo[] pi = m.GetParameters(); //Получить параметры.
            if(m.Name.CompareTo("MyMethod")==0 )
            {
                m.Invoke(reflectOb, args);
            }
        }
    }
}
```

Метод GetConstructors()

Метод GetConstructors() возвращает массив объектов класса ConstructorInfo, описывающих конструкторы.

Метод GetConstructors()

Метод GetConstructors() возвращает массив объектов класса ConstructorInfo, описывающих конструкторы.

Форма метода Invoke() для создания объекта

```
object Invoke(object[] parameters)
```

Метод GetConstructors()

Метод GetConstructors() возвращает массив объектов класса ConstructorInfo, описывающих конструкторы.

Форма метода Invoke() для создания объекта

```
object Invoke(object[] parameters)
```

Пример:
```
Type t = typeof(MyClass);
ConstructorInfo[] ci = t.GetConstructors();
object reflectOb = ci[2].Invoke(1,1);
```

Метод GetConstructors()

Метод GetConstructors() возвращает массив объектов класса ConstructorInfo, описывающих конструкторы.

Форма метода Invoke() для создания объекта

```
object Invoke(object[] parameters)
```

Пример:
```
Type t = typeof(MyClass);
ConstructorInfo[] ci = t.GetConstructors();
object reflectOb = ci[2].Invoke(1,1);
```

Упражнение 9.11

Создать объект с помощью рефлексии.