Расширенные возможности полиморфизма в языке C#

© Учебный Центр безопасности информационных технологий Microsoft Московского инженерно-физического института (государственного университета), 2003

Содержание лекции

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Основные результаты исследований полиморфизма

1934 – А. Черч (Alonso Church) изобрел ламбда-исчисление и исследовал порядок вычислений в ламбда-термах

1936 – Г. Плоткин (G.D. Plotkin) исследовал стратегии вызова по имени и по значению на основе ламбда-исчисления

1960's – П.Лендин (Peter J. Landin) создал SECD-машину, математический формализм, моделирующий вызов по имени

1969 – Р.Хиндли (Roger Hindley) исследовал полиморфные системы с типами

1978 –Р.Милнер (Robin Milner) предложил расширенную систему полиморфной типизации для языка программирования ML

1989-90 – У.Кук, П.Кэннинг, У.Хилл и др. (William R. Cook, Peter S. Canning, Walter L. Hill et al.) исследовали полиморфизм в ООП и его связь с ламбда-исчислением

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Понятие полиморфизма в программировании

Вообще говоря, под ***полиморфизмом*** понимается возможность оперировать объектами, не обладая точным знанием их типов.

В функциональном программировании и ООП понятие полиморфизма связано с:

- наследованием;
- интерфейсами;
- отложенным связыванием (“ленивыми” или “замороженными” вычислениями)

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Полиморфизм типов в языке SML

Встроенная функция `hd` для списка произвольного типа:

```
hd [1, 2, 3];
val it = 1: int
```
(тип функции: (int list) → int)

```
hd [true, false, true, false];
val it = true: bool
```
(тип: (bool list) → bool)

```
hd [(1,2)(3,4),(5,6)];
val it = (1,2) : int*int
```
((int*int)list→(int*int))

Функция `hd` имеет тип (*type* `list`) →*type*, где *type* – произвольный тип

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Полиморфизм типов в языке C#

Рассмотрим пример полиморфной функции:

```
void Poly(object o) {
  Console.WriteLine(o.ToString());
}
```

а также примеры ее применения:

```
Poly(25);
Poly("John Smith");
Poly(3.141592536m);
Poly(new Point(12,45));
```

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Абстрактные классы в языке C#

1.Средство реализации концепции полиморфизма.

2.Абстрактные методы не имеют части реализации (implementation).

3. Абстрактные методы неявно являются виртуальными (virtual).

4. В случае, если класс имеет абстрактные методы, его необходимо описывать как абстрактный.

5. Запрещено создавать объекты абстрактных классов.

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Абстрактные классы в языке C#: пример применения

```
abstract class Stream {
  public abstract void Write(char ch);
  public void WriteString(string s) {
        foreach (char ch in s) Write(s);
 }
}

class File : Stream {
  public override void Write(char ch) {
        ... write ch to disk ...
  }
}
```

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Абстрактные свойства и методы в языке C# (пример)

```
    abstract class Sequence {
    public abstract void Add(object x);    // метод
    public abstract string Name { get; }   // свойство
    public abstract object this [int i] {get;set;}
                                           // индексатор
    }

class List : Sequence {
      public override void Add(object x) {...}
      public override string Name { get {...} }
      public override object this [int i] { get {...} set
    {...} }
    }
```

© Microsoft Information Security and Technology Training Center
Moscow Engineering Physics Institute (State University), 2003

«Запечатанные» классы в языке C#

«Запечатанными» (sealed) называют нерасширяемые классы, которые могут наследовать свойства других классов.

Замещенные (`override`) методы могут описываться как `sealed` в индивидуальном порядке.

Такого рода решения возможны по следующим соображениям:

1) безопасность (предотвращается изменение семантики класса);
2) эффективность (методы могут вызываться путем статического связывания)

```
sealed class Account : Asset {
  long val;
  public void Deposit (long x) { ... }
  public void Withdraw (long x) { ... }
  ...
}
```

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Основы динамического связывания в языке C# (1)

```
class A {
  public virtual void WhoAreYou() {
Console.WriteLine("I am an A"); }
}

class B : A {
  public override void WhoAreYou() {
Console.WriteLine("I am a B"); }
}
```

Сообщение запускает метод, который динамически определяет принадлежность к классу

```
A a = new B();
a.WhoAreYou();                    // "I am a B"
```

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Основы динамического связывания в языке C# (2)

Каждый из методов, который способен обрабатывать объект класса `A`, способен также обрабатывать объект класса `B` (см. следующий пример):

```
void Use (A x) {
  x.WhoAreYou();
}

Use(new A()); // " I am an A"
Use(new B()); // " I am a B"
```

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Сокрытие в языке C#

Экземпляры могут быть описаны как `new` в подклассе.

Они скрывают одноименные наследуемые экземпляры.

```
class A {
  public int x;
  public void F() {...}
  public virtual void G() {...}
}
class B : A {
  public new int x;
  public new void F() {...}
  public new void G() {...}
}
B b = new B();
b.x = ...;                          // имеет доступ к B.x
b.F(); ... b.G();                   // вызывает B.F и B.G
((A)b).x = ...;                     // имеет доступ к A.x
((A)b).F(); ... ((A)b).G();         // вызывает A.F и A.G
```

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Сложное (с сокрытием) динамическое связывание в языке C# (1)

```
class A {
public virtual void WhoAreYou(){
  Console.WriteLine("I am an A");
  }
}

class B : A {
public override void WhoAreYou(){
  Console.WriteLine("I am a B");
  }
}

class C : B {
public new virtual void WhoAreYou() {
  Console.WriteLine("I am a C");
  }
}
```

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Сложное (с сокрытием) динамическое связывание в языке C# (2)

```
class D : C {
public override void WhoAreYou(){
  Console.WriteLine("I am a D");
  }
}

C c = new D();
c.WhoAreYou();                    // "I am a D"

A a = new D();
a.WhoAreYou();                    // "I am a B"
```

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Требования к методам с приоритетами:

1. Идентичность описаний:
- одинаковые количество и типы параметров (включая типы функций);
- одинаковые области видимости (public, protected, …).

2. Свойства и индексаторы также могут иметь приоритет (virtual, override).

3. Статические методы не могут иметь приоритета.

4. Только методы, описанные как виртуальные, могут иметь приоритет в производных классах.

5. Методы с приоритетами необходимо описывать как `override`

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Пример применения методов с приоритетами

```
class A {
  public void F() {...}   // может иметь приоритет
  public virtual    void G() {...}
                // может иметь приоритет в подклассе
}

class B : A {
  public      void F() {...}
// предупреждение: скрывается производный метод F() ->
необходимо использовать оператор new
  public      void G() {...}
// предупреждение: скрывается производный метод G() ->
необходимо использовать оператор new
  public override   void G() {
        // ok: перекрывает приоритетом производный G
        ... base.G(); // вызов производного G()
  }
}
```

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Понятие интерфейса в языке C#

Под ***интерфейсом*** понимается чисто абстрактный класс, содержащий только описания без реализации.

Свойства интерфейсов:

1) могут содержать методы, свойства, индексаторы и события (но не поля, константы, конструкторы, деструкторы, операторы и вложенные типы);
2) все элементы являются общедоступными и абстрактными (виртуальными);
3) ни один из элементов не может быть статическим;
4) множественные наследования могут реализовываться классами и структурами;
5) интерфейсы могут расширяться другими интерфейсами

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Пример интерфейса

Рассмотрим следующий пример интерфейса:

```
public interface IList : ICollection, IEnumerable {
  int Add (object value);
                                        // методы
  bool Contains (object value);
  ...
  bool IsReadOnly { get; }
                                        // свойство
  ...
  object this [int index] { get; set; }
                                        // индексатор
}
```

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Реализация интерфейса на основе классов и структур

Сформулируем требования к реализации интерфейсов:

1. Класс может наследовать свойства единственного базового класса и при этом реализовывать множественные интерфейсы.
2. Структура может наследовать свойства любого типа и при этом реализовывать множественные интерфейсы.
3. Каждый элемент интерфейса (метод, свойство, индексатор) может реализовать или наследовать свойства класса.
4. Реализованные методы интерфейса нельзя описывать как `override`.
5. Реализованные методы интерфейса можно описывать как `virtual` или `abstract` (т.е. интерфейс может быть реализован посредством абстрактного класса).

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Пример реализации интерфейса на основе классов и структур

Рассмотрим следующий пример реализации интерфейса на основе классов и структур:

```
class MyClass : MyBaseClass, IList, ISerializable {
      public int Add (object value) {...}
      public bool Contains (object value) {...}
      ...
      public bool IsReadOnly { get {...} }
      ...
      public object this [int index] {
            get {...}
            set {...}
      }
}
```

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Преимущества концепции полиморфизма

1. Унификация обработки объектов различной природы
2. Снижение стоимости программного обеспечения
3. Повторное использование кода
4. Интуитивная прозрачность исходного текста
4. Строгое математическое основание (ламбда-исчисление)
5. Концепция является универсальной и в равной степени применима в функциональном и объектно-ориентированном программировании

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Библиография (1)

1. Pratt T.W., Zelkovitz M.V. Programming languages, design and implementation (4th ed.).- Prentice Hall, 2000
2. Appleby D., VandeKopple J.J. Programming languages, paradigm and practice (2nd ed.).- McGraw-Hill, 1997
3. Milner R. A theory of type polymorphism in programming languages. Journal of Computer and System Science, 17(3):348-375, 1978
4. Canning P.S., Cook W.R., Hill W.L., Olthoff W., Mitchell J.C. F-bounded polymorphism for object-oriented programming.Conference on Functional Programming and Computer Architecture, 1989, p.p. 273-280

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003

Библиография (2)

5. Thorup L., Tofte M. Object-oriented programming and Standard ML. Proc. ACM SIGPLAN 1994 Workshop on ML and its applications, Orlando, FL, June 1994, Tech. Report 2265 INRIA, p.p. 41-49
6. Troelsen A. C# and the .NET platform (2nd ed.).- APress, 2003, 1200 p.p.
7. Liberty J. Programming C# (2nd ed.).- O'Reilly, 2002, 656 p.p.
8. Plotkin G.D. Call-by-name, call-by-value and the λ-calculus. Theoretical computer science, 1, pp. 125-159, 1936
9. Turner D.A. A new implementation technique for applicative languages. Software – Practice and Experience, 9:21-49, 1979

© Учебный Центр безопасности информационных технологий Microsoft
Московского инженерно-физического института (государственного университета), 2003