Расширенные возможности полиморфизма в языке C#

Комментарий к слайду

В данной лекции будут рассмотрены вопросы, относящиеся к истории развития, идеологии, математическому основанию и обзору расширенных возможностей полиморфизма – одной из фундаментальных концепций, на которых основано объектно-ориентированное программирование.

Содержание лекции

Комментарий к слайду

Коротко о содержании лекции.

В ходе лекции будут рассмотрены важнейшие научные исследования, относящиеся к эволюции подходов к математическому моделированию расширенных возможностей одного из важнейших для объектно-ориентированного подхода к программированию концепций аспекта, а именно, полиморфизма.

Прежде всего, будет представлен сравнительный анализ путей реализации абстрактных структуры данных в языках SML и C#.

Затем будут рассмотрены так называемые виртуальные классы и методы, а также способы представления абстрактных методов, свойств и индексаторов в языке программирования C#.

При этом будут подробно исследованы особенности реализации так называемых «запечатанных» (sealed) классов.

Кроме того, будет произведено сопоставление вариантов связывания объектов языка программирования C# с их значениями для динамического и статического случаев.

Особое внимание будет уделено реализации механизмов абстрактных классов в языке программирования C# на основе использования механизма интерфейсов, причем будет рассмотрено применение интерфейсов на основе классов и структур.

Фрагменты программ на языках программирования SML и C# проиллюстрируют практику использования концепции полиморфизма и дадут возможность сопоставления особенностей ее реализации в зависимости от подхода.

Лекция завершится обзором литературы для более глубокого исследования материала.

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

Комментарий к слайду

Кратко остановимся на наиболее значительных (с точки зрения данного курса) этапах эволюции теории и практики реализации полиморфизма в формальных теориях и языках программирования. Заметим, что рассматриваемая проблематика не раз затрагивалась нами в ходе изложения курса благодаря тому обстоятельству, что она является основополагающей для современного computer science.

В 30-х г.г. А. Черч (Alonso Church) предложил исчисление ламбда-конверсий или ламбда-исчисление и применил его для исследования теории множеств. Вклад ученого был настолько фундаментальным, что теория (до сих пор называемая ламбда-исчислением и часто именуемая в литературе ламбда-исчислением Черча) является основополагающей и для рассматриваемых нами вопросов.

Исследования различных стратегий передачи параметров при обращении к (полиморфным) функциям языков программирования (в частности, вызова функций по имени и по значению) на основе ламбда-исчисления были проведены Г. Плоткиным (G.D. Plotkin). Заметим, что полученные результаты значительно позднее (уже в 70-х г.г.) были использованы для моделирования вычислений в языках программирования.

В 60-х г.г., уже в эпоху высокоуровневых языков программирования, П. Лендином (Peter J. Landin) была разработана так называемая SECD-машина, а именно, математическая формализация для реализации вычислений в терминах ламбда-выражений. Кроме того, тем же автором был создан формальный язык, представляющий собой вариант расширенного ламбда-исчисления и ставший впоследствии прообразом ряда языков программирования.

В конце 60-х г.г. Р. Хиндли (Roger Hindley) исследовал полиморфные системы типов, т.е. такие системы типов, в которых возможны параметризованные функции или функции, имеющие переменный тип. При этом основной проблемой, стоящей перед исследователем, было моделирование языков программирования со строгой типизацией.

Затем, в 70-х г.г. Р. Милнер (Robin Milner) предложил практическую реализацию расширенной системы полиморфной типизации для языков программирования.

Наконец, на рубеже 80-х и 90-х г.г., рядом исследователей – У.Куком (William R. Cook), П.Кэннингом (Peter S. Canning), У.Хиллом (Walter L. Hill) и другими – была изучена концепция полиморфизма в приложении к объектно-ориентированному программированию (в частности, к языку C++) и выявлена возможность моделирования полиморфизма в ООП на основе ламбда-исчисления.

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)

Комментарий к слайду

По рассмотрении теоретических оснований такой важнейшей концепции объектно-ориентированного программирования как полиморфизм, предпримем попытку формализовать это понятие.

Под полиморфизмом будем иметь в виду возможность оперирования объектами без однозначной идентификации их типов.

Напомним, что понятие полиморфизма уже исследовалось в части курса, посвященной функциональному подходу к программированию.

Наметим концепции, объединяющие функциональный и объектно-ориентированный подходы к программированию с точки зрения полиморфизма.

Как было отмечено в ходе исследования функционального подхода к программированию, концепция полиморфизма предполагает в части реализации отложенное связывание переменных со значениями. При этом во время выполнения программы происходят так называемые “ленивые” или, иначе, “замороженные” вычисления. Таким образом, означивание языковых идентификаторов выполняется по мере необходимости.

В случае объектно-ориентированного подхода к программированию теоретический и практический интерес при исследовании концепции полиморфизма представляет отношение наследования, прежде всего, в том отношении, что это отношение порождает семейства полиморфных языковых объектов.

С точки зрения практической реализации концепции полиморфизма в языке программирования C# в форме полиморфных функций особую значимость для исследования представляет механизм интерфейсов.

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

Комментарий к слайду

Напомним, что реализация полиморфизма при функциональном подходе к программированию основана на оперировании функциями переменного типа.

Для иллюстрации исследуем поведение встроенной функции `hd`, (от слова «head» – голова), которая выделяет «голову» (первый элемент) списка, вне зависимости от типа его элементов. Применим функцию к списку из целочисленных элементов:

```
hd [1, 2, 3];
val it = 1: int
```

Получим, что функция имеет тип функции из списка целочисленных величин в целое число (`int list → int`). В случае списка из значений истинности та же самая функция

```
hd [true, false, true, false];
val it = true: bool
```

возвращает значение истинности, т.е. имеет следующий тип: `bool list → bool`.
Наконец, для случая списка кортежей из пар целых чисел

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

получим тип `((int*int)list→ (int*int))`. В итоге можно сделать вывод о том, что функция `hd` имеет тип (*type* `list`) → *type*, где *type* – произвольный тип, т.е. полиморфна.

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));
```


Комментарий к слайду

Проиллюстрируем сходные черты и особенности реализации концепции полиморфизма при функциональном и объектно-ориентированном подходе к программированию следующим примером фрагмента программы на языке C#:

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

Как видно, приведенный пример представляет собой описание полиморфной функции `Poly`, которая выводит на устройство вывода (например, на экран) произвольный объект `o`, преобразованный к строковому формату (`o.ToString()`).

Рассмотрим ряд примеров применения функции `Poly`:

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

Заметим, что независимо от типа аргумента (в первом случае это целое число `25`, во втором – символьная строка `"John Smith"`, в третьем – вещественное число π=`3.141592536`, в четвертом – объект типа `Point`, т.е. точка на плоскости с координатами `(12,45)`), обработка происходит единообразно и, как и в случае с языком функционального программирования SML, функция генерирует корректный результат.

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.

Комментарий к слайду

Исследовав особенности реализации различных стратегий вычислений в языке программирования C#, рассмотрим концепцию полиморфизма в соотнесении с механизмом так называемых абстрактных классов.

Абстрактные классы при объектно-ориентированном подходе (в частности, в языке программирования C#) являются аналогами полиморфных функций в языках функционального программирования (в частности, в языке SML) и используются для реализации концепции полиморфизма. Методы, которые реализуют абстрактные классы, также называются абстрактными и являются полиморфными.

Перечислим основные особенности, которыми обладают абстрактные классы и методы в рамках объектно-ориентированного подхода к программированию.

Прежде всего, отметим то обстоятельство, что абстрактные методы не имеют части реализации (implementation).

Кроме того, абстрактные методы неявно являются виртуальными (т.е. как бы оснащенными описателем `virtual`).

В том случае, если внутри класса имеются определения абстрактных методов, данный класс необходимо описывать как абстрактный. Ограничений на количество методов внутри абстрактного класса в языке программирования C# не существует.

Наконец, в языке программирования C# запрещено создание объектов абстрактных классов (как конкретизаций или экземпляров).

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 ...
  }
}
```


Комментарий к слайду

Проиллюстрируем особенности использования абстрактных классов следующим фрагментом программы на языке C#:

```
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 ...
          }
     }
```

Как видно из приведенного примера, фрагмент программы на языке C# представляет собой описание абстрактных классов `Stream` и `File`, реализующих потоковую запись (метод `Write`) данных в форме символьных строк `ch`.

Заметим, что описание абстрактного класса `Stream` реализовано явно посредством зарезервированного слова `abstract`. Оператор `foreach … in` реализует последовательную обработку элементов.

Необходимо обратить внимание на то обстоятельство, что поскольку в производных классах необходимо замещение методов, метод `Write` класса `File` оснащен описателем `override`.

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
{...} }
}
```


Комментарий к слайду

Проиллюстрируем особенности использования абстрактных свойств и методов следующим фрагментом программы на языке C#:

```
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
     {...} }
}
```

Как видно из приведенного примера, фрагмент программы на языке C# представляет собой описание абстрактного класса `Sequence` и `List`, реализующих потоковое чтение и запись данных (`get` и `set`).

Заметим, что описание абстрактного класса `Sequence` реализовано явно посредством зарезервированного слова `abstract`.

Необходимо обратить внимание на то обстоятельство, что поскольку в производных классах необходимо замещение методов, методы `Add` и `Name` класса `List` оснащены описателем `override`.

«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) { ... }
  ...
}
```


Комментарий к слайду

Еще одним средством реализации расширенного полиморфизма в языке программирования C# является механизм, известный под названием «запечатанных» (sealed) классов.

Под «запечатанными» классами будем понимать нерасширяемые классы, которые могут наследовать свойства других классов. Решение об использовании механизма «запечатывания» при описании приоритетных или, иначе, замещенных (`override`) методов принимается в индивидуальном порядке.

Рассмотрим соображения, обосновывающие использование механизма «запечатанных» классов в языке программирования C#. Прежде всего, благодаря реализации данного механизма предотвращается изменение семантики класса. Таким образом существенно повышается безопасность разрабатываемого программного обеспечения. Кроме того, за счет того, что вызов «запечатанных» методов предполагает использование статического связывания, значительно возрастает эффективность реализуемых приложений.

Проиллюстрируем особенности использования механизма «запечатанных» классов следующим фрагментом программы на языке C#:

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

Как видно из приведенного примера, фрагмент программы на языке C# представляет собой описание «запечатанного» класса `Account`, реализующего абстрактные методы `Deposit` и `Withdraw` для занесения средств на счет и их снятия со счета.

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"
```


Комментарий к слайду

В ходе курса уже обсуждалась стратегия динамического связывания переменных со значениями при функциональном и объектно-ориентированном подходах.

Проиллюстрируем особенности реализации динамического связывания в языке программирования C# следующим фрагментом программы:

```
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"); }
}
```

Как видно из приведенного примера, данный фрагмент программы содержит описания класса `A` с виртуальным методом `WhoAreYou`, который выводит на стандартное устройство вывода сообщение о принадлежности к классу `A`, а также замещенного класса `B` с виртуальным методом `WhoAreYou`, который выводит на стандартное устройство вывода сообщение о принадлежности к классу `B`. При таком подходе сообщение запускает метод, который динамически определяет принадлежность к классу (в приведенном ниже примере создается и идентифицируется объект класса `B`):

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

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"
```


Комментарий к слайду

Заметим, что существенной особенностью рассматриваемого примера является то обстоятельство, что B – это замещенный метод.

В этой связи, каждый из методов, который способен обрабатывать A, способен также обрабатывать B. Проиллюстрируем этот факт следующим фрагментом программы на языке C#:

```
void Use (A x) {
     x.WhoAreYou();
}
Use(new A());  // " I am an A"
Use(new B());  // " I am a B"
```

Как видно из приведенного фрагмента программы, применение метода Use, реализующего идентификацию объекта класса A, приводит к результату для объекта " I am an A" класса A и к результату " I am a B" для объекта класса B. Корректность получаемого результата для класса B (несмотря на детерминированное описание класса A как параметра метода Use) объясняется тем обстоятельством, что класс B является замещенным.

Преимущества концепции полиморфизма

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

Комментарий к слайду

Подводя промежуточные итоги рассмотрения основных аспектов расширенных возможностей концепции полиморфизма в объектно-ориентированном подходе к программированию и особенностей реализации этой концепции применительно к языку программирования C#, кратко отметим достоинства полиморфизма.

Прежде всего, к преимуществам концепции полиморфизма следует отнести унификацию обработки объектов различной природы. В самом деле, абстрактные классы и методы позволяют единообразно оперировать гетерогенными данными, причем для адаптации к новым классам и типам данных не требуется реализации дополнительного программного кода.

Кроме того, важным практическим следствием реализации концепции полиморфизма для экономики программирования является снижение стоимости проектирования и реализации программного обеспечения.

Еще одним существенным достоинством полиморфизма является возможность усовершенствования стратегии повторного использования кода. Код с более высоким уровнем абстракции не требует существенной модификации при адаптации к изменившимся условиям задачи или новым типам данных.

Важно также отметить, что идеология полиморфизма основана на строгом математическом фундаменте (в частности, в виде формальной системы ламбда-исчисления), что обеспечивает интуитивную прозрачность исходного текста для математически мыслящего программиста, а также верифицируемость программного кода.

Наконец, концепция полиморфизма является достаточно универсальной и в равной степени применима для различных подходов к программированию, включая функциональный и объектно-ориентированный.

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
```


Комментарий к слайду

Рассмотрим особенности реализации сокрытия данных в языке программирования C#.

Заметим, что в языке C# существует возможность описания объектов как экземпляров в составе подкласса с использованием зарезервированного слова new. При этом происходит сокрытие одноименных наследуемых экземпляров в подклассах.

Проиллюстрируем особенности использования механизма сокрытия данных в языке программирования C# следующим фрагментом программы:

```
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
```

Как видно из приведенного фрагмента программы, базовый класс A и производный класс B характеризуются общедоступными полем x и методами F и G. Особенности доступа к элементам классов приведены в комментариях. Заметим, что при описании элементов производного класса используется зарезервированное слово new.

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");
  }
}
```


Комментарий к слайду

Рассмотрим особенности реализации сложного динамического связывания в языке программирования C#. Под сложным связыванием будем понимать связывание с сокрытием (hiding) данных. Для детальной иллюстрации особенностей использования механизма сокрытия данных воспользуемся следующим фрагментом программы в качестве развернутого примера на языке C#. Приведем начало примера:

```
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");
     }
}
```

Как видно из первой части примера, фрагмент программы содержит описания общедоступных классов: базового класса `A` и производных классов `B` и `C` с иерархией `C` ISA `B` ISA `A`. Каждый из классов характеризуется единственным общедоступным методом `WhoAreYou` для вывода в стандартный поток диагностического сообщения о принадлежности к данному классу. Заметим, что методы `WhoAreYou` для классов `A` и `C` описаны как виртуальные, причем в последнем описании используется зарезервированное слово `new`. Метод `WhoAreYou` для класса `B` описан как замещенный.

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"
```


Комментарий к слайду

Продолжим изложение развернутого примера, демонстрирующего сложное динамическое связывание в языке программирования C#. Приведем окончание примера на языке C#:

```
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"
```

Как видно из заключительной части примера, фрагмент программы содержит описание общедоступного класса `D` как производного от `C`.

Таким образом, иерархия классов принимает вид: `D` ISA `C` ISA `B` ISA `A`. Класс `D` характеризуется аналогичным предыдущим классам общедоступным методом `WhoAreYou` для вывода в стандартный поток диагностического сообщения о принадлежности к данному классу. Заметим, что метод `WhoAreYou` для класса `D` является замещенным (но не виртуальным), и в его описании не используется зарезервированное слово `new`.

При инициализации объектов `c` и `a` как экземпляров класса `D`, применение «отладочных» методов дает результат `"I am a D"` для объекта `c` и результат `"I am a B"` для объекта `a`. Полученный результат объясняется расположением описателей методов `override` в иерархии классов (более верхний описатель имеет более высокий приоритет).

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`

Комментарий к слайду

Дальнейшим развитием динамического связывания в языке программирования C# является реализация механизма вызова методов с приоритетами.

При реализации рассматриваемого механизма на взаимосвязанные семейства методов накладывается ряд дополнительных ограничений.

Прежде всего, требуется обеспечить идентичность описаний методов в группе с приоритетами. При этом для каждого из методов должны выполняться следующие условия.

Во-первых, у всех методов семейства для вызова с приоритетами должны быть одинаковыми как количество параметров, так и типы этих параметров. Типы функций не должны составлять исключение из этого правила. Во-вторых, все методы семейства для вызова с приоритетами должны иметь одинаковые области видимости (например, каждый из методов семейства должен иметь описатель `public` или, скажем, `protected`; однако, наличие в семействе методов с разными описателями области видимости не допускается).

Кроме того, необходимо отметить, что свойства и индексаторы для семейства методов с приоритетами также могут иметь приоритет (с использованием описателя `override`). Поскольку метод с приоритетами является принципиально динамическим объектом языка программирования, статические методы не могут описываться как методы с приоритетами.

Заметим, что только методы, описанные как виртуальные (`virtual`), могут иметь приоритет в производных классах.

Наконец, при задании методов с приоритетами необходимо использовать описатель `override`.

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()
  }
}
```


Комментарий к слайду

Проиллюстрируем особенности использования механизма методов с приоритетами в языке программирования C# следующим фрагментом программы:

```
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()
     }
}
```

Как видно из приведенного примера, фрагмент программы содержит описания базового класса `A` и производного класса `B`, каждый из которых содержит общедоступные методы `F` и `G`. При этом метод `G` класса использует описатель `virtual`. Как следует из комментариев, при задании методов `F` и `G` в производном классе `B` без использования описателя `override` происходит сокрытие производных методов `F` и `G`. Таким образом, для корректной работы `F` и `G` в классе необходимо использовать оператор `new`. В случае использования описателя `override` при задании метода `G` в классе `B`, происходит замещение приоритетным методом `G` и, таким образом, оператор `new` не требуется.

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.

Комментарий к слайду

В ходе изложения курса неоднократно упоминалось понятие интерфейса. На данном этапе исследований становится возможным произвести детальное рассмотрение данного механизма и рассмотреть особенности его реализации применительно к языку программирования C#.

Под интерфейсом будем понимать чисто абстрактный класс, который содержит только описания языковых объектов и не содержит части, отвечающей за реализацию (implementation).

При реализации рассматриваемого механизма необходимо обеспечить выполнение следующих условий.

Прежде всего, требуется, чтобы в состав интерфейсов входили только такие объекты языка как методы, свойства, индексаторы и события. Интерфейсы не могут содержать полей, констант, конструкторов, деструкторов, операторов и вложенных типов.

Кроме того, все элементы интерфейсов должны быть описаны как общедоступные (`public`) и абстрактные виртуальные (`virtual`).

В состав интерфейсов не могут входить статические элементы.

Механизм множественного наследования, рассмотренный ранее в ходе курса, может быть реализован посредством классов и структур (далее будет рассмотрен еще один пример подобного рода наследования).

Заметим также, что произвольный интерфейс может быть расширен посредством одного или нескольких интерфейсов.

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
}
```


Комментарий к слайду

Проиллюстрируем особенности использования механизма методов с приоритетами в языке программирования C# следующим фрагментом программы:

```
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
}
```

Как видно из приведенного примера, фрагмент программы на языке C# содержит описание общедоступного интерфейса `IList` на основе стандартных классов `ICollection` и `IEnumerable`. Интерфейс моделирует списковую структуру данных и оснащен методами `Add` для добавления объекта в список, `Contains` для установления принадлежности объекта списку и `IsReadOnly` для определения возможности записи в список (а, возможно, и рядом других методов), тем или иным набором свойств, а также индексатором.

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).

Комментарий к слайду

Рассмотрим более подробно особенности реализации множественного наследования в языке программирования C#. Для достижения этой цели потребуется применить механизм интерфейсов для классов и структур.

Сформулируем требования, необходимые для реализации данного варианта механизма интерфейсов.

Прежде всего, следует потребовать, чтобы класс имел возможность наследования свойств лишь единственного базового класса. Однако, при этом класс должен иметь возможность реализации множественных интерфейсов.

Кроме того, необходимо выдвинуть условия для структур, которые сводятся к следующему. Структура может наследовать свойства любого типа данных и при этом имеет возможность реализации множественных интерфейсов.

Подчеркнем, что важным свойством элементов, входящих в состав интерфейса (в частности, методов, свойств и индексаторов) является возможность реализации или наследования свойств порождающего их класса.

Заметим далее, что реализованные методы интерфейса не могут быть замещенными, т.е. задаваться с помощью описателя `override`.

Наконец, методы, которые реализуются посредством интерфейса, могут снабжаться одним из описателей `virtual` либо `abstract`. Таким образом, осуществляется возможность реализации механизма интерфейсов на основе абстрактного класса.

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 {...}
      }
}
```


Комментарий к слайду

Проиллюстрируем особенности использования механизма методов с приоритетами в языке программирования C# следующим фрагментом программы:

```
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 {...}
     }
}
```

Как видно из приведенного примера, фрагмент программы на языке C# содержит описание общедоступного интерфейса `IList` на основе стандартных классов `ICollection` и `IEnumerable`.

Данный пример во многом схож с предыдущим, однако имеет следующее фундаментальное отличие от него.

Отличие заключается в том, что интерфейс `IList` в данном примере реализуется на основе некоторого базового класса `MyBaseClass`, определенного пользователем. При этом интерфейс `IList` в данном примере фактически получает возможность множественного наследования.

Заметим, что ни один из методов интерфейса `Ilist` не является замещенным.

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

Комментарий к слайду

К сожалению, в рамках времени, отведенных на одну лекцию, можно лишь в общих чертах охарактеризовать специфику такой фундаментальной концепции объектно-ориентированных языков, ООП, а также для целого ряда других подходов к программированию как полиморфизм. Для более детального ознакомления с особенностями, достижениями и проблемами в теории моделирования этой концепции и практики реализации связанных с ней механизмов рекомендуется следующий список литературы:

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

Кратко остановимся на источниках. В работе [1] приведен наиболее полный анализ истории развития и особенностей языков программирования с классификацией по областям применения. В работе [2] рассмотрены вопросы проектирования и реализации современных языков программирования. Работа [3] представляет собой исследование концепции полиморфизма в условиях функционального подхода к программированию. Работа [4] посвящена исследованию применимости полиморфизма в функциональном программировании к ООП.

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

Комментарий к слайду

Продолжим обсуждение работ, посвященных исследованию теоретических основ и практической реализации концепции полиморфизма.

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

Работа [5] анализирует взаимосвязи полиморфизма в рамках объектно-ориентированного и функционального подходов к программированию и иллюстрирует их примерами программ на языках SML и C++. В работе [6] рассмотрены теоретические проблемы и практические аспекты реализации инновационных конструкций в языках программирования, прежде всего, в языке C#.NET, изучение которого составляет основу курса. В работе [7] рассматриваются вопросы, связанные с разработкой программ на языке C#. Работа [8] представляет собой сравнительное исследование стратегий вычислений в современном программировании. Работа [9] посвящена формализации полиморфизма средствами теоретического computer science.