Основы объектной модели .NET

Класс object и его методы

Для базового класса `System.Object` в C# можно использовать его синоним `object`. Методы класса `object` имеются у всех типов .NET Framework:

- `public virtual string ToString();`
- `public Type GetType();`
- `public static bool ReferenceEquals(object o1, object o2);`
- `public static bool Equals(object o1, object o2);`
- `public virtual bool Equals(object o2);`
- `public virtual int GetHashCode().`

Размерные типы (value type)

Определяются с использованием описателя `struct` и называются *структурами*. Частный случай размерных типов — *перечисления* (описатель `enum`). Структурами являются все числовые типы, тип `char` и тип `bool`.

Особенности размерных типов (структур):

- структуры не поддерживают наследования (за исключением возможности *упаковки*),
- структуры не могут содержать описания конструктора без параметров (хотя он всегда доступен). Использование конструктора для создания структур так же обязательно, как и для классов,
- структуры не могут содержать финализаторы и виртуальные методы,
- для полей структуры нельзя использовать инициализаторы,
- в явно определяемых конструкторах необходимо задавать значения всех полей.

Пример: описание структуры с тремя ошибками, которое станет правильным, если заменить `struct` на `class`:

```
public struct Point
{
  int x = 1;
  int y;
  public Point() {}
  public Point(int x) { this.x = x; }
}
```

Nullable-структуры и операции ??, ?. и ?[]

Nullable-структуры — это структуры с возможностью хранения «неопределенного» (indefined) значения — `null`. Они реализуются с помощью обобщенной структуры `Nullable<T>`, но имеется упрощенный синтаксис: `int?`, `bool?`, `Point?`. Преобразование `vtype` → `vtype?` выполняется по умолчанию, обратное преобразование требует *приведения типа* (cast), например, `i = (int) inull`

Операция `??`: `i = inull ?? 0`. Никогда не приводит к возбуждению исключения. Может применяться не только к Nullable-типам, но и к обычным ссылочным типам.

В версии C# 6.0 (2015 год) появилась *null-условная операция* `?.` и `?[]`. Если объект `a` имеет значение `null`, то попытка обращения к его полям, свойствам и методам (например, `a.x`) приводит к возбуждению исключения `NullReferenceException`. Если же использовать null-условную операцию (`a?.x`), то в аналогичной ситуации будет просто возвращено значение `null` (даже если требуемое поле или свойство имеет ссылочный тип или метод возвращает ссылочное значение). Попытка вызова `void`-метода с применением null-условной операции для объекта `null` просто игнорируется. Операция `?[]` аналогичным образом обрабатывает обращения к элементам массивов и других классов, имеющих свойства-*индексаторы*.

Упаковка и распаковка данных размерного типа

При присваивании объекта размерного типа переменной, которая имеет тип `object` или интерфейсный тип (в частности, при передаче такого объекта в качестве object-параметра), выполняется *упаковка* этого объекта: Обратное действие называется *распаковкой*. Упаковка выполняется неявно, распаковка требует явного преобразования. При упаковке всегда создается *копия* исходного объекта в куче.

Пример:

```
int i = 10;
object o = i; // упаковка целого числа
int j = (int)o;
double d = (double)o; // ошибка времени выполнения
double d = (int)o; // d = 10.0
o = 5.7 // упаковка вещественного числа
int k = (int)(double)o; // k = 5
```

Более содержательный пример упаковки:

```
ArrayList L = new ArrayList();
L.Add(10); // метод Add описан как Add(object value)
int i = (int)L[0];
```

По возможности упаковки следует избегать. Иногда это можно сделать простой модификацией исходного кода. В последующих примерах `i` — переменная типа `int`, `o` — переменная типа `object`, содержащая упакованное целое.

```
Console.WriteLine(i + ", " + (int)o); // 2 упаковки и 1 распаковка
Console.WriteLine(i + ", " + o); // 1 упаковка
Console.WriteLine(i.ToString() + ", " + o); // 0 упаковок
Console.WriteLine(i); // 0 упаковок
```

Модификаторы доступа

Для *типов*:

- `internal` (по умолчанию), `public`:,
- `abstract`;
- `sealed`;
- `static`.

Для *членов класса*: предусмотрены основные модификаторы `private` (по умолчанию), `protected`, `internal`, `public`.

Дополнительно для *полей*:

- `static`;
- `readonly`;
- `const` (допустимо указывать только для данных числовых типов, `bool`, `char`, `string`, а также перечислимых типов).

Дополнительно для *методов*:

- `static`;
- `virtual`;
- `override`;
- `new`;
- `abstract`;
- `sealed`.

Перегрузка методов и конструкторы

Перегруженные методы имеют одинаковые имена, но их списки параметров должны различаться; в частности, для разных перегруженных методов могут использоваться параметры, имеющие и не имеющие модификатор `ref`, например, `int x` и `ref int x`.

Конструкторы экземпляров: в отличие от остальных членов класса *не наследуются*. Если ни один конструктор не определен, то создается конструктор без параметров. После заголовка конструктора можно указывать вариант вызова конструктора предка (`: base(…)`) или другого конструктора этого же типа (`: this(…)`). Класс может содержать *статический конструктор*, который используется для инициализации статических членов и выполнения действий, общих для класса в целом. Всегда описывается с атрибутом `static`, не имеет параметров и всегда является закрытым (хотя атрибут `private` для него не указывается).

Свойства

Свойства (properties) выглядят как поля, но обрабатываются с помощью методов.

Свойства без параметров:

```
string name;
public string Name
{
  get { return name; }
  set { name = value; }
// value – имя параметра метода get,
// тип параметра value совпадает с типом свойства
}
```

Методы get и set (*аксессоры*) автоматически генерируются компилятором. Пример:

```
a.Name += "***";
// a.set_Name(a.get_Name() + "***");
```

Более содержательный пример свойства:

```
class Button
{
  …
  int width;
  public Width
  {
    get { return width; }
    set
    {
      if (value == width)
        return;
      // если новое значение совпадает со старым, то никакие действия не выполняются
      if (value <= 0)
        throw new ArgumentOutOfRangeException
          ("<текст сообщения об ошибке>");
          // обработка ошибочного нового значения
      width = value;
      Redraw(); // выполнение побочного действия
    }
  }
  …
}
```

Свойства с параметрами (*индексаторы*):

```
string[] a;
public string this[int index]
{
  get { return a[index]; }
  set { a[index] = value; }
}
```

Пример: индексатор класса string, возвращающий символ с требуемым индексом: s[i] (данный индексатор является свойством только для чтения, в то время как аналогичный индексатор класса StringBuilder доступен как для чтения, так и для записи). Индекс не обязан быть целочисленным (в отличие от индексов массива). Индексов может быть несколько; допускается перегрузка индексаторов.

Быстрое определение свойства, доступного *извне* только для чтения (эта возможность появилась в C# 3.0):

```
public int Name
{ get; private set; }
```

В версии C# 6.0 количество синтаксического сахара для свойств было увеличено. Примеры новых возможностей:

```
public string Name
{
  get => name;
  set => name = value; // лямбда-синтаксис при определении методов (не только аксессоров!)
}
public string Name { get; set; } = "***"; // инициализация автосвойства
public string Name { get; } = "***"; // только для чтения (в том числе и внутри класса)
```

Инициализаторы объектов, анонимные типы и описатель var (C# 3.0)

Сразу после вызова конструктора можно указать в фигурных скобках *инициализаторы* для требуемых полей или свойств.

Пример:

```
public class Bunny
{
  public string Name;
  public bool LikesCarrots;

  public Bunny() {}
  public Bunny(string n) { Name = n; }
}
…
Bunny b1 = new Bunny { Name = "Bo", LikesCarrots = true };
Bunny b2 = new Bunny("Bo") { LikesCarrots = true };
```

В версии C# 6.0 аналогичный синтаксис распространен на инициализаторы элементов *словарей*:

```
Dictionary<string, int> d = new Dictionary<string, int> { ["A"] = 1, ["A"] = 2, ["A"] = 3 };
```

Анонимные типы — безымянные классы, создаваемые «на лету» и содержащие набор свойств *только для чтения* (подобным образом можно создавать только *локальные* переменные):

```
var b3 = new { Name = "Bo", LikesCarrots = true };
b3.Name += "*"; // ошибка компиляции
```

Имя поля анонимного типа может быть получено из имени инициализированной переменной:

```
string Name = "Bo";
var b4 = new { Name, LikesCarrots = true };
Console.WriteLine(b4.Name); // Bo
```

В C# 7.0 аналогичная (и даже более общая) функциональность может быть достигнута с помощью *кортежей* (тип `System.ValueTuple`, в отличие от типа `System.Tuple`, введенного ранее и существенно менее удобного):

```
var b5 = (Name : "Bo", LikesCarrots : true ); // тип (string Name, bool LikesCarrots);
b5.Name += "*"; // допустимо: кортежи являются изменяемыми!
(string, bool) b6 = b5; // допустимо, хотя поля кортежа b6 являются неименованными
Console.WriteLine(b6.Item1); // Bo*
(string s1, bool p1) = b6; // деконструкция кортежа, s1 = "Bo*", p1 = true
(s1, p1) = b6; // деконструкция кортежа при присваивании
(_, p1) = b6; // частичная деконструкция кортежа при присваивании
var b7 = (s1, p1); // можно обращаться к полям так: b7.Item1 или b7.s1, b7.Item2 или b7.p1
Console.WriteLine(b7 == b6); // true, при сравнении (C# 7.3) имена полей не учитываются
```

Делегаты и события

Делегат — аналог процедурного типа. Реализуется компилятором в виде потомка класса `System.MulticastDelegate`, однако C# позволяет обойтись без явного описания класса:

```
public delegate int Transformer(int i);
// новый тип делегата с сигнатурой int (int i)
```

Пример использования:

```
public int Triple(int i)
  { return 3*i; }
…
Transformer t = Triple;
```

В первой версии C# приходилось вызывать *конструктор делегата* (начиная с версии 2.0 это является необязательным):

```
t = new Transformer(Triple);
```

В случае экземплярных методов делегату передается метод вместе с именем экземпляра:

```
t = a.Triple;
```

В случае статических методов для метода указывается имя класса:

```
t = Math.Abs;
```

Все делегаты могут *группировать методы* (эта способность определена в классе `MulticastDelegate`). Пример:

```
public delegate void InfoDelegate(string s);
class Informer
{
  public static InfoDelegate Info;
  public static void NewInfo(string s)
  {
    if (Info != null)
      Info(s);
  }
// этот метод вызывается, когда классу Informer надо проинформировать своих "подписчиков";
// если не выполнять проверку, то при отсутствии подписчиков будет возбуждено исключение
}
class A
{
  public string InfoLog;
  public void GetInfo(string s)
  { InfoLog += s + ". "; }
}
class B
{
  public static string InfoLog;
  public static void GetInfo(string s)
  { InfoLog += s + ". "; }
}
class Program
{
  static void Main(string[] args)
  {
    Informer.NewInfo("Event0"); // подписчиков нет
    A a = new A();
    Informer.Info += a.GetInfo;
    Informer.NewInfo("Event1");
    Console.WriteLine(a.InfoLog); // Event1.
    Console.WriteLine(B.InfoLog); // пустая строка
    Informer.Info += B.GetInfo;
    Informer.NewInfo("Event2");
    Console.WriteLine(a.InfoLog); // Event1. Event2.
    Console.WriteLine(B.InfoLog); // Event2.
```

```
    Informer.Info -= a.GetInfo;
    Informer.Info -= a.GetInfo;
      // при повторном отказе от подписки
      // не выполняется никаких действий
    Informer.NewInfo("Event3");
    Console.WriteLine(a.InfoLog); // Event1. Event2.
    Console.WriteLine(B.InfoLog); // Event2. Event3.
    Console.ReadLine();
  }
}
```

Если значение делегата равно `null`, то операция += эквивалентна обычной инициализации =.

В приведенной схеме отсутствует защита от возможных попыток одного из подписчиков вмешаться в действия информатора, связанные с другими подписчиками:

- `Informer.Info = B.GetInfo;`
- `Informer.Info = null;`
- `Informer.Info("False event")` (имитация события).

События (events) формализуют рассмотренную ранее схему обмена сообщениями между источником широковещательных сообщений и его подписчиками, повышая ее надежность. Основные понятия, связанные с функционированием событий:

- *источник* — это тип, содержащий делегат, связанный с некоторым событием; определяет, в какой момент надо вызывать этот делегат, инициируя соответствующее событие;
- *подписчик* — это тип, получающий от источника информацию о событии посредством своего метода, присоединенного к этому событию (*обработчика события*); подписчик определяет (с помощью операций += и -=), когда начинать и заканчивать *прослушивание события*;
- *событие* — это фактически оболочка для делегата, оставляющая открытой только ту часть его функциональных возможностей, которые необходимы для модели «источник–подписчик».

Пример:

```
public static event InfoDelegate Info;
```

Если теперь попытаться выполнить любое из «запрещенных» действий, описанных ранее, возникнет ошибка компиляции. В том классе, в котором определено событие, с ним можно работать как с обычным делегатом.

Модифицируем предыдущий пример, используя стандартную модель событий, применяемую в стандартных классах библиотеки .NET. Кроме того, сделаем событие не классовым, а экземплярным.

```
public class InfoEventArgs : EventArgs
{
  public readonly string Info;
  public InfoEventArgs(string s)
  { Info = s; }
}
public delegate void InfoEventHandler(object sender, InfoEventArgs e);
public class Informer
{
  public event InfoEventHandler Info;
  protected virtual void OnInfo(InfoEventArgs e) // диспетчер события Info
  {
    if (Info != null)
      Info(this, e);
  }
  public void NewInfo(string s) // метод, имитирующий наступление события
  {
    OnInfo(new InfoEventArgs(s));
  }
```

```
  public string Id;
}
class A
{
  public string InfoLog;
  public void GetInfo(object sender, InfoEventArgs e)
  {
    InfoLog += string.Format("From {0}: {1}. ", (sender as Informer).Id, e.Info);
  }
}
class B
{
  public static string InfoLog;
  public static void GetInfo(object sender, InfoEventArgs e)
  {
    InfoLog += string.Format("From {0}: {1}. ", (sender as Informer).Id, e.Info);
  }
}
class Program
{
  static void Main(string[] args)
  {
    Informer i1 = new Informer { Id = "i1"},
      i2 = new Informer { Id = "i2"};
    A a = new A();
    i1.Info += a.GetInfo;
    i2.Info += a.GetInfo;
    i1.NewInfo("Event1");
    i2.NewInfo("Event2");
    Console.WriteLine(a.InfoLog); // From i1: Event1. From i2: Event2.
    Console.WriteLine(B.InfoLog); // пустая строка
    i2. Info -= a.GetInfo;
    i2.Info += B.GetInfo;
    i1.NewInfo("Event3");
    i2.NewInfo("Event4");
    Console.WriteLine(a.InfoLog); // From i1: Event1. From i2: Event2. From i1: Event3.
    Console.WriteLine(B.InfoLog); // From i2: Event4.
    Console.ReadLine();
  }
}
```

Действия компилятора при обработке события:

- создается закрытый делегат;
- определяются открытая пара методов, обеспечивающих применение операций += и -= к закрытому делегату;
- любое обращение к событию в пределах содержащего его класса переводится в обращение к соответствующему закрытому делегату.

В C# можно реализовать нестандартные действия при подключении или отключении обработчика:

```
  InfoEventHandler info; // закрытый делегат
  public event InfoEventHandler Info; // открытое событие с явно определенными методами
  {
    add { info += value; }
    remove { info -= value; }
```

```
}
```

В этом примере явная реализация на самом деле работает аналогично реализации, которую по умолчанию обеспечивает компилятор. Однако теперь при добавлении или удалении обработчиков мы можем выполнять дополнительные действия, например, возбуждать специализированные события или исключения.

Если событие не несет дополнительной информации, а важен лишь факт его наступления (и, возможно, данные о его отправителе), то вместо определения нового делегата можно использовать стандартный делегат `EventHandler`:

```
public delegate void EventHandler(object sender, EventArgs e);
```

Операции `+=` и `-=` могут приводить к созданию нового объекта-делегата. В частности, если связать с делегатом `a` функцию `f`, затем выполнить присваивание `b = a` (в результате метод `object.ReferenceEquals(a, b)` вернет значение `true`), после чего добавить к делегату `a` функцию `g` (выполнив присваивание `a += g`), то делегат `b` *не изменится* (он по-прежнему будет связан только с функцией `f`), а метод `object.ReferenceEquals(a, b)` вернет значение `false`. Если после этого выполнить присваивание `a -= g`, то метод `object.ReferenceEquals(a, b)` опять вернет значение `true`.

Анонимные методы и лямбда-выражения

Пример использования *анонимного метода* (C# 2.0):

```
public delegate int Transformer(int i);
…
Transformer t = delegate(int i) { return 3 * i; };
t(4); // 12
```

Если параметры в анонимном методе не используются, то их можно не указывать (вместе с круглыми скобками):

```
button1.Click += delegate{ MessageBox.Show("…"); };
```

В C# 3.0 концепция анонимных методов получила дальнейшее развитие в *лямбда-выражениях*.

Варианты определения лямбда-выражения:

```
Transformer t = (int i) => {return 3 * i;};
Transformer t = (int i) => 3 * i;
Transformer t = i => 3 * i;
```

Синтаксис лямбда-выражения:

параметры => выражение_или_операторный_блок

Пример с использованием *внешних переменных*:

```
delegate int IntSequence();
static void Main(string[] args)
{
  int n = 0;
  IntSequence s = () => n++;
  // или IntSequence s = delegate { return n++; };
  Console.WriteLine(s());  //  0
  Console.WriteLine(s());  //  1
  Console.ReadLine();
}
```

При включении внешней переменной в анонимный метод или лямбда-выражение она *захватывается*, т. е. ее время жизни продлевается до времени жизни соответствующего анонимного метода или лямбда-выражения. Модификация предыдущего примера:

```
static IntSequence Seq()
{
  int n = 0;
  return () => n++;
}
static void Main(string[] args)
{
  IntSequence s = Seq();
```

```
    Console.WriteLine(s()); //  0
    Console.WriteLine(s()); //  1
    Console.ReadLine();
  }
```

Методы расширения (C# 3.0)

Методы расширения (extensions) позволяют дополнить существующий класс новыми методами, не изменяя определение этого класса. Всегда описываются в виде статических методов вспомогательного статического класса, причем первый параметр должен снабжаться специальным модификатором `this`.

Пример:

```
public static class ExtDemo
{
  public static bool Odd(this int n)
  {
    return n % 2 != 0;
  }
  public static int CharCount(this string s, params char[] charSet)
  {
    int n = 0;
    foreach (char c in s)
      if (charSet.Contains(c))
        n++;
    return n;
  }
}
…
Console.WriteLine(5.Odd()); // True
Console.WriteLine("123.123.456".CharCount('1', '2', '5')); // 5
```

Еще один пример:

```
public static class ExtSplit
{
  public static string[] Split(this string src, string separator, bool removeEmptyEntries)
  {
    return src.Split(new string[] { separator }, removeEmptyEntries ?
      StringSplitOptions.RemoveEmptyEntries : StringSplitOptions.None);
  }
  public static string[] Split(this string src, string separator)
  {
    return Split(src, separator, true);
  }
}
```

Если в классе определен метод экземпляра с тем же именем, что и метод расширения, то предпочтение всегда отдается методу экземпляра (метод расширения в подобной ситуации можно вызывать только как статический метод класса, в котором этот метод определен).