Об’єктно-зорієнтоване програмування на C#

Класс – це абстрактний тип даних. З допомогою класу описується деяка сутність (її характеристики і можливі дії). Наприклад, клас може описувати студента, автомобіль і т. д. Описавши клас, ми можемо створити його екземпляр – **об’єкт**. Об’єкт – це вже конкретний представник класу.

ОЗП

Інкапсуляція – дозволяє приховувати внутрішню реалізацію.

Успадкування – дозволяє створювати новий клас на базі іншого.

Поліморфізм – це здатність об'єктів з одним інтерфейсом мати різну реалізацію.

Абстракція – дозволяє виділяти з деякої сутності тільки потрібні характеристики і методи, які в повній мірі (для поставленої задачі) описують об'єкт.

План

1. Класи та об’єкти. Інкапсуляція.
2. Методи. Статичні Методи.
3. Конструктори.
4. Властивості, аксесори, автоматичні властивості.
5. Успадкування.
6. Масив вказівників на базовий клас. Оператори is, as.
7. Поліморфізм. Віртуальні методи. Перевизначення методів.
8. Абстрактні класи, методи, властивості.
9. Інтерфейси. Множинне успадкування.
10. Перевизначення операторів.

План

11. Перевизначення операторів.
12. Ссилочні типи та типи значень. Ref, Out
13. Значення NULL. Nullable-типы. Оператор ??.
14. Узагальнені типи даних, класи та методи.
15. Колекції: списки, стек, черга, словники. Індексатори і створення колекцій
16. Перерахування (Enum). Структури.
17. Перевизначення методів Equals, GetHashCode. Equals та “==”.
18. Регулярні вирази. Клас Regex.
19. Форматування рядків.
20. Делегати. Події. Лямбда.

Класи та об’єкти. Інкапсуляція

[модифікатор доступу] class [имя_класса] {

  //тело класса

}

збірки, на яку є посилання; Модифікаторів доступу для класів є два:

- *public* – доступ до класу можливий з будь-якого місця одній складання або з іншої
- *internal* – доступ до класу можливий тільки з збірки, в якій він оголошений.

Що таке збірка?

Збірка (assembly) – це готовий функціональний модуль у вигляді exe або dll-файлу (файлів), який містить скомпільований код .NET. Збірка надає можливість повторного використання коду.

Класи та об’єкти. Інкапсуляція

```
namespace HelloWorld
{
class Student //без модифікатору доступу, клас буде internal
   {
     //тіло класу
   }
}
```

Класи та об’єкти. Інкапсуляція

Члени класу

- *поля;*
- *константи;*
- *властивості;*
- *конструктори;*
- *методи;*
- *події;*
- *оператори;*
- *індексатори;*
- *вкладені типи.*

Класи та об’єкти. Інкапсуляція

Всі члени класу, як і сам клас, мають свій рівень доступу. Тільки у членів їх може бути вже п'ять:

- *public* – доступ до члена можливий з будь-якого місця однієї збірки, або з іншої збірки, на яку є посилання;
- *protected* – доступ до члена можливий тільки усередині класу, або в класі-спадкоємці (при спадкуванні)
- *internal* – доступ до члена можливий тільки з збірки, в якій він оголошений;
- *private* – доступ до члена можливий тільки усередині класу;
- *protected internal* - доступ до члена можливий з однієї збірки, або з класу-спадкоємця іншої збірки.

Класи та об’єкти. Інкапсуляція

Пример объявления полей в классе:

```
class Student
{
   private string firstName;
   private string lastName;
   private int age;
   public string group; // не рекомендується використовувати public для
поля
}
```

Класи та об’єкти. Інкапсуляція

```
namespace HelloWorld {
  class Student  {
    private string firstName;   private string lastName;
    private int age;   public string group;
  }
  class Program {
    static void Main(string[] args)  {
      Student student1 = new Student(); //створення об'єкта student1 класу Student
      Student student2 = new Student();
    } } }
```

Класи та об’єкти. Інкапсуляція

```
static void Main(string[] args)
{
  Student student1 = new Student();
  Student student2 = new Student();
  student1.group = "Group1";
  student2.group = "Group2";
  Console.WriteLine(student1.group); // виводить на екран "Group1"
  Console.Write(student2.group);
  Console.ReadKey();
}
```

Класи та об’єкти. Інкапсуляція

```
static void Main(string[] args)
{
   Student student1 = new Student();
   student1.firstName= "Nikolay"; //помилка немає доступу до поля
firstName. Программа не скомпілюється
}
// константи
class Math1 {
   private const double Pi = 3.14;
}
```

Класи та об’єкти. Інкапсуляція

```
Book b1 = new Book();

b1.name = "Война и мир";

b1.author = "Л. Н. Толстой";

b1.year = 1869;
```

Инициализаторы об'єктів

```
Book b2 = new Book { name = "Отцы и дети",

author = "И. С. Тургенев", year = 1862 };
```

Методи

Метод – це невелика підпрограма, яка виконує, в ідеалі, тільки одну функцію.

Статичний метод – це метод, який не має доступу до полів об'єкта, і для виклику такого методу не потрібно створювати екземпляр (об'єкт) класу, в якому він оголошений.

Простий метод – це метод, який має доступ до даних об'єкта, і його виклик виконується через об'єкт.

Методи

```
class TVSet {

   private bool switchedOn;

   public void SwitchOn()  {

     switchedOn = true;

}

   public void SwitchOff() {

     switchedOn = false;

}  }

class Program {
```

Методи

```
class StringHelper {
   public static string TrimIt(string s, int max)
{
     if (s.Length <= max)
       return s;
     return s.Substring(0, max) + "...";
   }
}

class Program {
   static void Main(string[] args)
   {
```

Методи

Статичний метод не має доступу до нестатическим полів класу:

```
class SomeClass {
   private int a;
   private static int b;
   public static void SomeMethod()
   {
     a=5; // помилка
     b=10; // допустимо
   }
}
```

Конструктори

Конструктор – це метод класу, призначений для ініціалізації об'єкта при його створенні.

Ім’я конструктора завжди збігається з ім'ям класу.

При **оголошенні** конструктора, не потрібно вказувати тип повертається.

Конструктор слід оголошувати як **public**.

Існує **неявний** конструктор за замовчуванням.

```
public [имя_класса] ([аргументи]) {
  // тіло конструктора
}
```

Конструктори

```
class Car {
   private double mileage;  private double fuel;
   public Car()  {    //оголошення конструктора
     mileage = 0; // пробіг
     fuel = 0; // паливо
   }
}
class Program {
   static void Main(string[] args) {
     Car newCar = new Car(); // створення об'єкта і виклик конструктора
   }
}
```

Конструктори

```
public Car(double mileage, double fuel) {// конструктор с параметрами
    this.mileage = mileage;
    this.fuel = fuel;
  }
 Car newCar = new Car(100, 50); //виклик конструктора з параметрами
```

Вказівник this - це покажчик на об'єкт, для якого був викликаний нестатический метод. Ключове слово this забезпечує доступ до поточного екземпляру класу.

Конструктори

У класі можливо вказувати безліч конструкторів, головне щоб вони відрізнялися сигнатурами. **Сигнатура**, у разі конструкторів, - це набір аргументів. Наприклад, можна створити два конструктора, які приймають два аргументи типу *int*.

Якщо в класі визначено один або кілька конструкторів з параметрами, ми не зможемо створити об'єкт через неявний конструктор за замовчуванням:

Конструктори

```
class Car  {
   private double mileage; private double fuel;
   public Car() {
     mileage = 0;  fuel = 0;
   }
  public Car(double mileage, double fuel) {
     this.mileage = mileage;
     this.fuel = fuel;
  } }
class Program {
```

Властивості

Властивість – це член класу, який надає механізм доступу до поля класу. При використанні властивості компілятор перетворює це звернення до викликом відповідного неявного методу. Такий метод називається аксессор (accessor). Існує два таких методи: get (для отримання даних) і set (для запису).

Оголошення простого властивості має наступну структуру:

```
[модифікатор доступу] [тип] [ім'я_властивості] {
  get  {  // тіло аксессора для читання з поля }
  set  {  // тіло аксессора для запису у полі }
}
```

Властивості

```
class Student {
  private int year;     //оголошення закритого поля
  public int Year  {     //оголошення властивості
    get  {    // аксессор читання поля
      return year;
    }
    set   {  // аксессор запису у полі
      if (value < 1)  year = 1;
      else if (value > 5)  year = 5;
      else year = value;
} } }
class Program  {
  static void Main(string[] args)  {
```

Властивості

```
class Student {
   private int year;
   public int GetYear()  {
     return year;
   }
   public void SetYear(int value) {
     if (value < 1)  year = 1;
     else if (value > 5)   year = 5;
     else year = value;
 }  }
class Program
```

Властивості

Необхідно закрити доступ на запис

```
class Student  {
   private int year;
   public Student(int y)   {   // конструктор
     year = y;
   }
   public int Year  {
     get  {
       return year;
} } }
class Program {
   static void Main(string[] args) {
```

Властивості

Автоматичні властивості

```
class Student {
   public int Year { get; set; }
}
class Program {
   static void Main(string[] args) {
     Student st1 = new Student();
     st1.Year = 0;
     Console.WriteLine(st1.Year);
     Console.ReadKey();
} }
```

```
public int Year { private get; set; } // властивість тільки на запис
```

Успадкування

Людина - > студент, вчитель, фермер, …

```
class Animal {
   public string Name { get; set; }
}
class Dog : Animal {
   public void Guard() {
     // собака охороняє
   }
}
class Cat : Animal {
   public void CatchMouse() {
     // кішка ловить мишу
} }
class Program {
```

Успадкування

Виклик конструктора базового класу

Коли конструктор визначено тільки в спадкоємця, то тут все просто – при створенні об'єкта спочатку викликається конструктор за замовчуванням базового класу, а потім конструктор спадкоємця.

Коли конструктори оголошені і в базовому класі, і в спадкоємця – нам необхідно викликати їх обидва. Для виклику конструктора базового класу використовується ключове слово base. Оголошення конструктора класу-спадкоємця з викликом конструктора базового має наступну структуру:

```
[имя_конструктора_класса-спадкоємця] ([аргументи]) : base ([аргументи]) {
  // тіло конструктора
}
```

Успадкування

```
class Animal {
   public string Name { get; set; }
   public Animal(string name) {
     Name = name;
}  }
class Parrot : Animal {
   public double BeakLength { get; set; } // довжина дзьоба
   public Parrot(string name, double beak) : base(name) {
     BeakLength = beak;
}  }
class Dog : Animal {
  public Dog(string name) : base (name)  {
```

Масив вказівників на базовий клас. Оператори is, as

В Сі-шарп є можливість створення **масиву (чи списку) покажчиків на базовий клас**, у якому в якості елементів можуть бути об'єкти класу-спадкоємця. Наприклад, ми можемо створити масив об'єктів Тварина, і елементами такого масиву будуть об'єкти класів Собака, Кішка. Приклад:

Масив вказівників на базовий клас. Оператори is, as

```
class Animal {
   public string Name { get; set; }
   public Animal(string name) {
     Name = name;
} }
class Dog : Animal {
   public Dog(string name) : base(name)  { }
   public void Guard() { // собака охороняє
} }
class Cat : Animal {
  public Cat(string name) : base(name) { }
  public void CatchMouse() { // кішка ловить мишу
} }
class Program {
```

Масив вказівників на базовий клас. Оператори is, as

Хоча як елементи в цей список ми додавали об'єкти класів-спадкоємців Собака і Кішка, будучи елементами списку покажчиків на базовий клас, ці об'єкти перетворюються на об'єктах базового класу, і ми маємо доступ тільки до тієї частини об'єктів, яка описана в базовому класі – ми не можемо тут викликати методи Guard() або CatchMouse(), але при цьому маємо доступ до імені тварини.

Зворотне тут неможливо. Не можна створити масив об'єктів класу Собака, та записати в нього об'єкти класу Тварина.

Масив вказівників на базовий клас. Оператори is, as

Оператор is

Оператор is працює дуже просто – він перевіряє сумісність об'єкта з зазначеним типом (належить об'єкт певного класу). Оператор is повертає істину (true), якщо об'єкт належить до класу. Істинна буде також при перевірці сумісності об'єкту класу-спадкоємця і базового класу:

```
foreach (Animal animal in animals) {
    if (animal is Dog) // перевіряємо чи є дана тварина собакою
      ((Dog)animal).Guard();
    else ((Cat)animal).CatchMouse();
  }
```

Масив вказівників на базовий клас. Оператори is, as

Оператор as

У наведеному вище прикладі, замість явного приведення типів можна було використовувати оператор as. (Dog)animal еквівалентно виразу animal as Dog. Різниця між оператором as і явним приведенням лише в тому, що в разі неможливості перетворення,оператор as повертає null, тоді як явне приведення викидає виключення.

```
foreach (Animal animal in animals) {
    if (animal is Dog) // перевіряємо чи є дана тварина собакою
      (animal as Dog).Guard();
    else (animal as Cat).CatchMouse();
}
```

Поліморфізм

Поліморфізм – це різна реалізація однотипних дій.
Віртуальний метод – це метод, який МОЖЕ бути перевизначено у класі-спадкоємці. Такий метод може мати стандартну реалізацію в базовому класі.
Абстрактний метод – це метод, який ПОВИНЕН бути реалізований у класі-спадкоємці. При цьому, абстрактний метод не може мати своєї реалізації в базовому класі (тіло пусте), на відміну від віртуального.

Перевизначення методу – це зміна реалізації методу, встановленого як віртуальний (в класі спадкоємця метод буде працювати відмінно від базового класу).

Поліморфізм

В якості системи, що надає той самий інтерфейс, в програмуванні може виступати клас і інтерфейс. Тут ми поговоримо про класах. Є клас, в ньому оголошено віртуальний або абстрактний метод. Від цього класу успадковуються ще кілька класів, і в кожному з них по-різному реалізується той самий віртуальний/абстрактний метод. Виходить, об'єкти цих класів мають метод з однаковим ім'ям, але з різною реалізацією. В цьому і є поліморфізм.

Віртуальні методи. Перевизначення методів

Віртуальний метод оголошується за допомогою ключового слова virtual:

```
[модифікатор доступу] virtual [тип] [ім'я методу] ([аргументи])  {
  // тіло методу
}
```

*Статичний метод не може бути віртуальним.

Оголосивши віртуальний метод, ми тепер можемо перевизначити його в класі спадкоємця. Для цього використовується ключове слово override:

```
[модифікатор доступу] override [тип] [ім'я методу] ([аргументи])  {
  // нове тіло методу
}
```

Віртуальні методи. Перевизначення методів

```
class Person  {
   public string Name { get; set; }
   public int Age { get; set; }
   public Person(string name, int age)
{ Name = name;  Age = age;  }
   public virtual void ShowInfo()  {
     Console.WriteLine("Человек\n
Имя: " + Name + "\n" + "Возраст: “
 + Age + "\n"); }  }
class Student : Person {
   public string HighSchoolName
{ get; set; }
public Student(string name, int age, string hsName): base(name, age)  {
     HighSchoolName = hsName;  }
public override void ShowInfo() {
     Console.WriteLine("Студент\n
Имя: " + Name + "\n" + "Возраст: "
```

Віртуальні методи. Перевизначення методів

Якщо відкинути ключові слова virtual і override? То в цьому випадку, в базовому класі і в класі спадкоємцеві методи з однаковим ім'ям ShowInfo. Програма працювати буде, але про кожному об'єкті, незалежно це просто людина або студент/учень, буде виводитися інформація тільки як про просту людину (буде викликатися метод ShowInfo з базового класу).Це можна виправити, додавши перевірки тип об'єкта, і за допомогою приведення типів, викликати потрібний метод ShowInfo:

```
foreach (Person p in persons)  {
   if (p is Student)   ((Student)p).ShowInfo();
   else if (p is Pupil)   ((Pupil)p).ShowInfo();
   else p.ShowInfo();
}
```

Віртуальні методи. Перевизначення методів

Буває так, що функціонал методу, який змінюється, в базовому класі мало відрізняється від функціоналу, який повинен бути визначений у класі спадкоємця. В такому випадку, при перевизначенні, ми можемо викликати спочатку цей метод базового класу, а далі дописати необхідний функціонал. Це робиться за допомогою ключового слова base:

```
public virtual void ShowInfo() {
// ShowInfo в классе Person
   Console.WriteLine("Имя: " + Name);
   Console.WriteLine("Возраст: " + Age); }
public override void ShowInfo()  {
// ShowInfo в классе Student
```

Абстрактні класи, методи та властивості

```
abstract class [имя_класса]  {
   //тело
}
```

Такий клас має такі особливості:

- не можна створювати екземпляри (об'єкти) абстрактного класу;
- абстрактний клас може містити як абстрактні методи/властивості, так і звичайні;
- у класі спадкоємці повинні бути реалізовані всі абстрактні методи і властивості, оголошені в базовому класі.

Абстрактні класи, методи та властивості

Абстрактний метод

[модифікатор доступу] abstract [тип] [ім'я методу] ([аргументи]);

Реалізація абстрактного методу в класі спадкоємця відбувається так само, як і перевизначення методу – за допомогою ключового слова override:

[модифікатор доступу] override [тип] [ім'я методу] ([аргументи])

```
{
  // реалізація методу
}
```

Абстрактні класи, методи та властивості

Абстрактні властивості

Створення абстрактних властивостей не сильно відрізняється від методів:

[модифікатор доступу] abstract [тип] [ім'я властивості] { get; set; }

Реалізація в класі-спадкоємці:

[модифікатор доступу] override [тип] [ім'я властивості]

```
{
  get { тіло аксессора get }
  set { тіло аксессора set }
}
```

```
abstract class Animal  {
   public string Name { get; set; }
   public string Type {get; set;}
   public abstract void GetInfo();  }
class Parrot : Animal  {
   public Parrot(string name)  {
     Name = name;  Type = "Птица"; }
   public override void GetInfo()   {
     Console.WriteLine("Тип:"+Type
 +"\n"+"Имя:"+Name+"\n");} }
class Cat : Animal  {
   public Cat(string name)  {
     Name = name;
     Type = "Млекопитающее"; }
   public override void GetInfo()  {
     Console.WriteLine("Тип:"+Type +"\n"+ "Имя:"+Name+"\n"); }  }
```

Інтерфейси. Множинне успадкування

Інтерфейси – це ще один інструмент реалізації поліморфізму в Сі-шарп. Інтерфейс являє собою набір методів (властивостей, подій, індексатори), реалізацію яких має забезпечити клас, який реалізує інтерфейс.

Інтерфейс користувача може містити тільки сигнатури (ім'я і типи параметрів) своїх членів. Інтерфейс не може містити конструктори, поля, константи, статичні члени. Створювати об'єкти інтерфейсу неможливо.

Оголошується за межами класу, за допомогою ключового слова interface:

```
interface ISomeInterface  {
   // тіло інтерфейсу
}
```

Інтерфейси. Множинне успадкування

Імена інтерфейсів прийнято давати, починаючи з префіксу «I», щоб відразу відрізняти де клас, а де інтерфейс.

Всередині інтерфейсу оголошуються сигнатури його членів, модифікатори доступу вказувати не потрібно:

```
interface ISomeInterface {
   string SomeProperty { get; set; } // властивість
   void SomeMethod(int a); // метод
}
```

Інтерфейси. Множинне успадкування

Клас, який реалізує інтерфейс, повинен надати реалізацію всіх членів інтерфейсу:

```
class SomeClass : ISomeInterface {
   public string SomeProperty {
     get  {  // тіло get аксессора  }
     set  {  // тіло set аксессора}
   }
   public void SomeMethod(int a) {
     // тіло методу }
}
```

Інтерфейси. Множинне успадкування

```
interface Igeometrical   {
   void GetPerimeter();
   void GetArea ();
}
class Rectangle : Igeometrical   {
   public void GetPerimeter()  {
     Console.WriteLine("(a+b)*2");
   }
   public void GetArea()  {
     Console.WriteLine("a*b");
} }


class Circle : Igeometrical   {
```

Інтерфейси. Множинне успадкування

Множинне спадкування є в мові C++ , а в C# від нього відмовилися і внесли інтерфейси. В C# клас може реалізувати відразу кілька інтерфейсів. Це і є головною відмінністю використання інтерфейсів і абстрактних класів. Крім того, звичайно ж, абстрактні класи можуть містити всі інші члени, яких не може бути в інтерфейсі, і не всі методи/властивості в абстрактному класі повинні бути абстрактними.

Інтерфейси. Множинне успадкування

```
interface IDrawable  {
   void Draw(); }
interface IGeometrical  {
   void GetPerimeter();
   void GetArea ();   }
class Rectangle : IGeometrical, IDrawable  {
   public void GetPerimeter()  {
      Console.WriteLine("(a+b)*2");  }
   public void GetArea()  {
      Console.WriteLine("a*b"); }
   public void Draw()  {
      Console.WriteLine("Rectangle");
} }
```

Перевантаження методів

Перевантаження методів – це оголошення у класі методів з однаковими іменами при цьому з різними параметрами.

Маючи певний метод, щоб його перевантажити, інший метод з таким же ім'ям повинен відрізнятися від нього кількістю параметрів і/або типами параметрів. Відмінності лише типами значень методами недостатньо для перевантаження, але якщо методи відрізняються параметрами, тоді перевантажуються методи можуть мати і різні типи значень.

Перевантаження методів

Приклад того, як може бути перевантажений метод:

```
public void SomeMethod() {
   // тіло методу
}
public void SomeMethod(int a) {
// від першого відрізняється наявністю параметра
   // тіло методу }
public void SomeMethod(string s) {
// від другого відрізняється типом параметра
   // тіло методу
}
public int SomeMethod(int a, int b) {
```

Перевантаження методів

Приклад того, як не може бути перевантажений метод:

```
public void SomeMethod(int a) {
   // тіло методу
}
public void SomeMethod(int b)   {   // імені параметра недостатньо
   // тіло методу
}
public int SomeMethod(int a)  {   // типу значення, що повертається недостатньо
   // тіло методу
   return 0;
}
```

Перевантаження методів

Метод *ToInt32()* може приймати параметр різного типу – *bool, float, double, byte, char*…

Приклад :

```
public static void AddAndDisplay(int a, int b)  {   Console.WriteLine(a + b);   }
public static void AddAndDisplay(char a, char b)  {  Console.WriteLine(a.ToString() + b.ToString());  }
static void Main(string[] args)  {
   AddAndDisplay(5, 8); // 13
   AddAndDisplay('C', '#'); // "C#"
   Console.ReadKey();
}
```

Перевантаження операторів

Перевантаження унарного* оператора:

```
public static [возвращаемый_тип] operator [оператор]([тип_операнда] [операнд])  {
  //функціонал оператора
}
```

Перевантаження бінарного* оператора:

```
public static [возвращаемый_тип] operator [оператор]([тип_операнда1] [операнд1],
[тип_операнда2] [операнд2]) {
  //функціонал оператора
}
```

Модифікатори public і static є обов'язковими.

Перевантаження операторів

Можна перевантажувати

Унарні оператори: +, -, !, ++, —, true, false

Бінарні оператори: +, -, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >=

Не можна перевантажувати

[] – функціонал цього оператора надають волонтери

() – функціонал цього оператора надають методи перетворення типів

+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>= короткі форми оператора присвоювання будуть автоматично доступні при перевантаженні відповідних операторів (+, -, * …).

Перевантаження операторів

```
public class Money {
  public decimal Amount { get; set; }
  public string Unit { get; set; }
  public Money(decimal amount, string unit) {
    Amount = amount; Unit = unit;
  }
  public static Money operator +(Money a, Money b) {
    if (a.Unit != b.Unit)
      throw new InvalidOperationException(“разные валюты");
    return new Money(a.Amount + b.Amount, a.Unit);
 } }
class Program {
  static void Main(string[] args)  {
```

Перегрузка операторов

```
public class Money  {
   public decimal Amount { get; set; }
   public string Unit { get; set; }
   public Money(decimal amount, string unit)  {
     Amount = amount;
     Unit = unit;
   }
   public static Money operator ++(Money a)  {
     a.Amount++;
     return a;
   }
   public static Money operator --(Money a)  {
```

Перевантаження операторів

Також існує можливість перевантаження самого операторного методу.

```
Public class Money {
  public decimal Amount { get; set; }
  public string Unit { get; set; }
  public Money(decimal amount, string unit) {
    Amount = amount;  Unit = unit;
  }
  public static Money operator +(Money a, Money b) {
    if (a.Unit != b.Unit)
      throw new InvalidOperationException(«Різні валюти");
    return new Money(a.Amount + b.Amount, a.Unit);
  }
  public static string operator +(string text, Money a) {
```

Вказівникові типи та типи значень. Ref та out

Типи значень

Цю категорію також називають структурними типами. Типи значень зберігаються на стеку. Дані змінної типу значення зберігаються в самій змінної. До типів значень відносяться:

- Цілочисельні типи (*byte, sbyte, char, short, ushort, int, uint, long, ulong*);
- Типи з плаваючою комою (*float, double*);
- Тип *decimal*;
- Тип *bool*;
- Користувальницькі структури (*struct*);
- Перерахування (*enum*).

Вказівникові типи та типи значень. Ref та out

Код нижче показує, що при присвоєнні значення однієї змінної значущого іншого типу, подальше зміна однієї із змінних не впливає на іншу. Так тому, що зберігання даних значущого типу відбувається в самій змінної:

```
int a = 1;
int b = 2;
b = a;
a = 3;
Console.WriteLine(a); // 3
Console.WriteLine(b); // 1
```

Вказівникові типи та типи значень. Ref та out

Змінна посилального типу містить не дані, а посилання на них. Самі дані у цьому випадку зберігаються в купі. Купа - це область пам'яті, в якій розміщуються керовані об'єкти, і працює збирач сміття. Збирач сміття звільняє всі ресурси і об'єкти, які вже не потрібні.

До посилальних типів відносяться:

- Класи (*class*);
- Інтерфейси (*interface*);
- Делегати (*delegate*);
- Тип *object*;
- Тип *string*.

Вказівникові типи та типи значень. Ref та out

У коді нижче був створений простий клас, в якому є одне поле типу int. Далі була прооблена така ж процедура, як і у випадку вище, тільки результат вже інший. Після присвоєння одного об'єкта до іншого, вони стали вказувати на одну і ту ж область пам'яті (міняємо b – змінюється і a):

```
class Test  {
   public int x;
}
class Program  {
   static void Main(string[] args)  {
     Test a = new Test();
     Test b = new Test();
     a.x = 1;
     b.x = 2;
```

Вказівникові типи та типи значень. Ref та out

В C# значення змінних по-замовчуванню передаються за значенням (метод передається локальна копія параметра, який використовується при виклику). Це означає, що ми не можемо всередині методу змінити параметр з поза:

```
public static void ChangeValue(int a)  {    a = 2;   }
static void Main(string[] args) {
   int a = 1;
   ChangeValue(a);
   Console.WriteLine(a); // 1
   Console.ReadLine(); }
```

Щоб передавати параметри за посиланням, і мати можливість впливати на зовнішню змінну, використовуються ключові слова ref out.

Вказівникові типи та типи значень. Ref та out

Щоб використовувати ref, це ключове слово варто вказати перед типом параметра в методі, і перед параметром при виклику методу:

```
public static void ChangeValue(ref int a)  {  a = 2; }
static void Main(string[] args) {
   int a = 1;  ChangeValue(ref a);
   Console.WriteLine(a); // 2    Console.ReadLine();
}
```

У цьому прикладі ми змінили значення зовнішньої змінної всередині методу. Особливістю ref є те, що змінна, яку ми передаємо в метод, обов'язково повинна бути проініціалізована значенням. Це є головною відмінністю ref від out.

Вказівникові типи та типи значень. Ref та out

```
public static void ChangeValue(out int a)  {
   a = 2;
}
static void Main(string[] args)  {
   int a;   ChangeValue(out a);
   Console.WriteLine(a); // 2
   Console.ReadLine();
}
```

Якщо не присвоїти нове значення параметру out, ми отримаємо помилку «out

The parameter 'a' must be assigned to before control leaves the current method»

Вказівникові типи та типи значень. Ref та out

Продуктивність

Враховуючи той факт, що за замовчуванням метод передаються параметри за значенням і створюються їх копії в стеку, при використанні складних типів даних (власні структури), або якщо метод викликається багато разів, це погано позначиться на продуктивності. В такому випадку також варто використовувати ключові слова ref і out.

Якщо говорити в цілому про посилальних типів і типи значень, то продуктивність програми впаде, якщо використовувати тільки посилальні типи. На створення змінної посилального типу в купі виділяється пам'ять під дані, а в стеку під посилання на ці дані. Для типів значень пам'ять виділяється тільки в стеку. Час на розміщення даних в стеку менше, ніж у купі, це також йде в плюс типами значень в

Значення Null. Nullable-типи. Оператор ??

Посилальні типи можуть приймати значення null, типи значень – ні.

Null вказує на те, що невідомо, чи, іншими словами, значення немає (не слід плутати 0 з null).

Object a = null; // нормально

int b = null; // ошибка, int не nullable тип

Nullable-типи

int? a = null; double? b = null; bool? c = null;

Оператор ?? null-об'єднання)

int? a = 1;

int? b = null;

Console.WriteLine(a ?? 3); // 1

Console.WriteLine(b ?? 3); // 3

Узагальнення

```
class Account {
    public int Id { get; set; }
    public int Sum { get; set; }
}
```

Тут ідентифікатор заданий як числове значення, тобто банківські рахунки будуть мати значення 1, 2, 3, 4 і так далі. Однак також нерідко для ідентифікатора використовуються і рядкові значення. І у числових, і у строкових значень є свої плюси і мінуси. І на момент написання класу ми можемо точно не знати, що краще вибрати для зберігання ідентифікатора - рядка або числа.

Узагальнення

І на перший погляд, щоб вийти з подібної ситуації, ми можемо визначити властивість Id як властивість типу object.

```
class Account  {
    public object Id { get; set; }  public int Sum { get; set; }
}
Account account1 = new Account { Sum = 5000 };
Account account2 = new Account { Sum = 4000 };
account1.Id = 2;    // упаковка в значення int до типу Object (продуктивність !!!)
account2.Id = "4356";
int id1 = (int)account1.Id;    // Розпакування в тип int (продуктивність !!!)
string id2 = (string)account2.Id;
```

Узагальнення

```
Account account2 = new Account { Sum = 4000 };
account2.Id = "4356";
int id2 = (int)account2.Id;     // Виняток InvalidCastException


class Account<T>
{
    public T Id { get; set; }
    public int Sum { get; set; }
}
```

Узагальнення

Кутові дужки в описі class Account<T> вказують, що клас є узагальненим, а тип T, укладений в кутові дужки, буде використовуватися цим класом. Необов'язково використовувати саме літеру T, це може бути і будь-яка інша буква або набір символів. Причому зараз нам невідомо, що це за тип, це може бути будь-який тип. Тому параметр T в кутових дужках ще називається універсальним параметром, так як замість нього можна підставити будь-який тип.

Узагальнення

```
Account<int> account1 = new Account<int> { Sum = 5000 };
Account<string> account2 = new Account<string> { Sum = 4000 };
account1.Id = 2;        // упаковка не потрібна
account2.Id = "4356";
int id1 = account1.Id;      // розпакування не потрібна
string id2 = account2.Id;
Account<string> account2 = new Account<string> { Sum = 4000 };
account2.Id = "4356";
int id1 = account2.Id;      // помилка компіляції
```

Узагальнення

Значення за замовчуванням. У цьому випадку нам треба використовувати оператор default(T). Він присвоює посилальних типів в якості значення null, а типами значень - значення 0:

```
class Account<T> {
    T id = default(T);
}
```

Узагальнення. Узагальнені методи

```
class Program {

    private static void Main(string[] args) {

        int x = 7;

        int y = 25;

        Swap<int>(ref x, ref y);

        Console.WriteLine($"x={x}    y={y}");

        string s1 = "hello";

        string s2 = "bye";

        Swap<string>(ref s1, ref s2);

        Console.WriteLine($"s1={s1}    s2={s2}");
```

Колекції

System.Collections (прості необобщенные класи колекцій), System.Collections.Generic (узагальнені або типізовані класи колекцій) и System.Collections.Specialized (спеціальні класи колекцій). Також для забезпечення паралельного виконання завдань і багатопотокового доступу застосовуються класи колекцій з простору імен System.Collections.Concurrent

Колекції

```
using System;
using System.Collections;
using System.Collections.Generic;
namespace Collections  {
    class Program  {
        static void Main(string[] args)  {
            // необобщенная коллекция ArrayList ArrayList objectList = new ArrayList() { 1, 2,
"string", 'c', 2.0f };
            object obj = 45.8;
            objectList.Add(obj);
            objectList.Add("string2");
objectList.RemoveAt(0); //видалення першого елемента
            foreach (object o in objectList)  {
                Console.WriteLine(o);
            }
Console.WriteLine("Загальна кількість елементів колекції: {0}", objectList.Count);
```

Колекції. Двухзв’язні списки

Value: саме значення вузла, представлене типом T

Next: посилання на наступний елемент типу LinkedListNode<T> у списку. Якщо наступний елемент відсутній, то має значення null

Previous: посилання на попередній елемент типуLinkedListNode<T> у списку. Якщо попередній елемент відсутній, то має значення null

Використовуючи методи класу LinkedList<T>, можна звертатися до різним елементам, як в кінці, так і на початку списку:

AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode): вставляє вузол newNode в список після вузла node.

AddAfter(LinkedListNode<T> node, T value): вставляє у списку новий вузол зі значенням value після вузла node.

Колекції. Двухзв’язні списки

AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode): вставляє в список вузол newNode перед вузлом node.

AddBefore(LinkedListNode<T> node, T value): вставляє у списку новий вузол зі значенням value перед вузлом node.

AddFirst(LinkedListNode<T> node): вставляє новий вузол в початок списку

AddFirst(T value): вставляє новий вузол зі значенням value в початок списку

AddLast(LinkedListNode<T> node): вставляє новий вузол в кінець списку

AddLast(T value): вставляє новий вузол зі значенням value в кінець списку

RemoveFirst(): видаляє перший вузол зі списку. Після цього новим першим вузлом стає вузол, наступного за вилученим

RemoveLast(): видаляє останній вузол зі списку

Колекції. Двухзв’язні списки

```
using System;
using System.Collections.Generic;
namespace Collections  {
   class Program  {
      static void Main(string[] args)   {
 LinkedList<int> numbers = new LinkedList<int>();
numbers.AddLast(1); // вставляємо вузол зі значенням 1 на останнє місце
 // так як в списку немає вузлів, то останнім буде також і першим
numbers.AddFirst(2); // вставляємо вузол зі значенням 2 на перше місце
numbers.AddAfter(numbers.Last, 3); // вставляємо після останнього вузла новий вузол із значенням 3
// тепер у нас список має наступну послідовність: 2, 1, 3
         foreach (int i in numbers) {
            Console.WriteLine(i);
         }
LinkedList<Person> persons = new LinkedList<Person>();
// додаємо persona в список і отримаємо об'єкт LinkedListNode<Person>, в якому зберігається ім'я Tom
LinkedListNode<Person> tom =
persons.AddLast(new Person() { Name = "Tom" });
```

Колекції. **Черга Queue<T>**

Клас Queue<T> представляє звичайну чергу, працює за алгоритмом FIFO ("перший увійшов - першим вийшов").

У класу Queue<T> можна відзначити наступні методи:

- Dequeue: витягує і повертає перший елемент черги
- Enqueue: додає елемент в кінець черги
- Peek: просто повертає перший елемент з початку черги без його видалення

Колекції. **Черга Queue<T>**

```
Queue<int> numbers = new Queue<int>();
        numbers.Enqueue(3); // чергу 3
        numbers.Enqueue(5); // чергу 3, 5
        numbers.Enqueue(8); // чергу 3, 5, 8
        // отримуємо перший елемент черги
        int queueElement = numbers.Dequeue();
//тепер черга 5, 8
        Console.WriteLine(numbers.Dequeue());
 Queue<Person> persons = new Queue<Person>();
 persons.Enqueue(new Person() { Name = "Tom" });
 persons.Enqueue(new Person() { Name = "Bill" });
 persons.Enqueue(new Person() { Name = "John" });
 // отримуємо перший елемент без його вилучення
        Person pp = persons.Peek();
        Console.WriteLine(pp.Name);


Console.WriteLine("Зараз в черзі{0}
```

Колекції. **Stack<T>**

Клас Stack<T> представляє колекцію, яка використовує алгоритм LIFO ("останній увійшов - першим вийшов"). При такій організації кожен наступний доданий елемент міститься поверх попереднього. Витяг з колекції відбувається в зворотному порядку - витягується той елемент, який знаходиться вище всіх в стеку.

У класі Stack можна виділити два основних методи, які дозволяють керувати елементами:

- Push: додає елемент в стек на перше місце
- Pop: витягує і повертає перший елемент стека
- Peek: просто повертає перший елемент стека без його видалення

Колекції. **Stack<T>**

```
Stack<Person> persons = new Stack<Person>();
        persons.Push(new Person() { Name = "Tom" });
        persons.Push(new Person() { Name = "Bill" });
        persons.Push(new Person() { Name = "John" });
        foreach (Person p in persons)
        {
            Console.WriteLine(p.Name);
        }
        // Перший елемент стеку
        Person person = persons.Pop(); // теперь в стеке Bill, Tom
        Console.WriteLine(person.Name);
```

Колекції. **Dictionary<T, V>**

```
Dictionary<int, string> countries = new Dictionary<int, string>(5);
countries.Add(1, "Russia");
countries.Add(3, "Great Britain");
countries.Add(2, "USA");
countries.Add(4, "France");
countries.Add(5, "China");
foreach (KeyValuePair<int, string> keyValue in countries)
{
    Console.WriteLine(keyValue.Key + " - " + keyValue.Value);
}
```

Колекції. **Dictionary<T, V>**

```
Dictionary<char, Person> people = new Dictionary<char, Person>();

people.Add('b', new Person() { Name = "Bill" });

people.Add('t', new Person() { Name = "Tom" });

people.Add('j', new Person() { Name = "John" });

foreach (KeyValuePair<char, Person> keyValue in people) {

// keyValue.Value представляє клас Person

Console.WriteLine(keyValue.Key + " - " + keyValue.Value.Name);

}

// перебір ключів

foreach (char c in people.Keys) {

    Console.WriteLine(c);

}
```

Колекції. **Dictionary<T, V>**

```
Dictionary<char, Person> people = new Dictionary<char, Person>();
people.Add('b', new Person() {
Name = "Bill" });
people['a'] = new Person() {
Name = "Alice" };
Dictionary<string, string> countries =
new Dictionary<string, string> {
    {"Франция", "Париж"},
    {"Германия", "Берлин"},
    {"Великобритания", "Лондон"}
};
```

Колекції. Індексатори і створення колекцій

Індексатори дозволяють індексувати об'єкти і використовувати їх як масиви. Фактично індексатори дозволяють нам створювати спеціальні сховища об'єктів або колекції. За формою вони нагадують властивості зі стандартними методами get і set, які повертають і привласнюють значення.

Колекції. Індексатори і створення колекцій

```
class Program {
   static void Main(string[] args)  {
      Library library = new Library();
      Console.WriteLine(library[0].Name);
      library[0] = new Book("Преступление и … ");
      Console.WriteLine(library[0].Name);
      Console.ReadLine();
 } }
class Book{
   public Book(string name) {
      this.Name=name;
   }
   public string Name { get; set; }
}
class Library{
   Book[] books;
   public Library() {
books = new Book[] { new Book("Отцы и дети"), new
```

Колекції. Індексатори і створення колекцій

```
class Matrix {
    private int[,] numbers = new int[,] { { 1, 2, 4}, { 2, 3, 6 }, { 3, 4, 8 } };
    public int this[int i, int j] {
        get  {
            return numbers[i,j];
        }
        set {
            numbers[i, j] = value;
} } }
Matrix matrix = new Matrix();
Console.WriteLine(matrix[0, 0]);
matrix[0, 0] = 111;
Console.WriteLine(matrix[0, 0]);
```

Перерахування (enum)

Перерахування (Enumeration) – це визначений користувачем цілочисельний тип, який дозволяє специфікувати набір допустимих значень, і призначити кожному зрозуміле ім'я.

Загальна структура: enum [имя_перечисления] { [имя1], [имя2], … };

приклад: enum Directions { Left, Right, Forward, Back };

Оголосивши таким чином перерахування, кожної символічно позначається константі присвоюється цілочисельне значення, починаючи з 0 (Left = 0, Right = 1 ...). Це цілочисельне значення можна задавати і самому:

enum Directions { Left, Right = 5, Forward = 10, Back };

Back у цьому прикладі буде мати значення 11.

Перерахування (enum)

```
enum Directions { Left, Right, Forward, Back };
// оголошення перерахування
class Program {
   public static void GoTo(Directions direction) {
     switch (direction) {
       case Directions.Back:
         Console.WriteLine("Go back");  break;
       case Directions.Forward:
         Console.WriteLine("Go forward");  break;
       case Directions.Left:
         Console.WriteLine("Turn left");   break;
       case Directions.Right:
         Console.WriteLine("Turn right ");   break;
```

Перерахування (enum)

Головні переваги, які нам дають перерахування це:

- Гарантія того, що змінним будуть призначатися допустимі значення вказаного набору;
- Коли ви пишете код програми в Visual Studio, завдяки засобу IntelliSense буде випадати список з допустимими значеннями, що дозволить заощадити деякий час, і нагадати, які можна використовувати;
- Код стає читабельний, коли в ньому присутні описові імена, а не ні про що не говорять числа.

Перерахування дуже широко використовуються в самій бібліотеці класів .NET. Наприклад, при створенні файлового потоку (FileStream) використовується перерахування FileAccess, за допомогою якого ми вказуємо з яким режимом доступу відкрити файл (читання/запис).

Структури

Структура – це більш проста версія класів. Всі структури успадковуються від базового класу *System.ValueType* і є типами значень, тоді як класи - посилальні типи. Структури оголошуються за допомогою ключового слова *struct*:

```
public struct Book {
   public string Name;  public string Year;    public string Author;
}
```

Примірник структури можна створювати без ключового слова *new*:

```
static void Main(string[] args) {
  Book b;    b. Name = "BookName";
}
```

Структури

Структури відрізняються від класів наступними речами:

- Структура не може мати конструктор без параметрів (конструктора за замовчуванням);
- Поля структури не можна ініціалізувати, крім випадків, коли статичні поля.

```
private int x = 0; // у структурі неприпустимо;
```

- Примірники структури можна створювати без ключового слова new;
- Структури не можуть успадковуватися від інших структур або класів. Класи не можуть успадковуватися від структур. Структури можуть реалізовувати інтерфейси;
- Так як структури це типи значень, вони володіють всіма властивостями подібних типів (передача в метод за значенням і т. д.), на відміну від посилальних типів;
- Структура може бути nullable типом.

Структури

Структури підходять для створення нескладних типів, таких як точка, колір, окружність. Якщо необхідно створити безліч екземплярів подібного типу, використовуючи структури, ми економимо пам'ять, яка могла б виділятися під посилання у випадку з класами.

Перевантаження методів Equals, GetHashCode. Equals та «==»

Всі класи є спадкоємцями базового класу *object*. У ньому є три віртуальних методу – *ToString*, *Equals* и *GetHashCode*. У цьому уроці ми поговоримо з вами про останніх двох методах, а також про оператора «==».

За замовчуванням при роботі з всіма класами крім *string*, інтерфейсами, делегатами) оператор «==» перевіряє рівність посилань. Він повертає true, коли обидві посилання вказують на один об'єкт, в іншому випадку – false. Наведу код, який демонструє роботу даного оператора з ссылочными типами:

Перевантаження методів Equals, GetHashCode. Equals та «==»

```
static void Main(string[] args) {
   object o1 = new object();
   object o2 = new object();
   object o3 = o1;
   Console.WriteLine(o1 == o2); // false
   Console.WriteLine(o1 == o3); // true
}
```

Тут створюється два об'єкта, посилання на які записуються в змінні o1 і o2. Далі посилання o1 копіюється в змінну o3 (o1 і o3 вказують на один об'єкт). У підсумку маємо false при порівнянні посилань o1 і o2, і true при o1 і o3.

Перевантаження методів Equals, GetHashCode. Equals та «==»

Метод *Equals* приймає один аргумент – об'єкт, який буде порівнюватися з поточним об'єктом, і визначає, чи рівні між собою ці об'єкти. Тут вже йде мова про рівність полів об'єктів, а не посилань. Цей метод віртуальний, та його базова реалізація це просто перевірка рівності посилань оператором «==». Але коли ми створюємо якийсь клас, і нам необхідно реалізувати можливість перевірки ідентичності об'єктів, слід перевизначити саме даний метод, а не скористатися перегрузкой оператора «==». При перевизначенні методу Equals слід подбати про те, щоб цей метод поверне false у випадках, коли метод передано значення NULL, коли переданий об'єкт не можна привести до типу поточного об'єкта, ну і коли поля об'єктів відрізняються.

Перевантаження методів Equals, GetHashCode. Equals та «==»

```
public class Money  {
   public decimal Amount { get; set; }
   public string Unit { get; set; }
   public Money(decimal amount, string unit)  {
     Amount = amount;   Unit = unit; }
  public bool Equals(Money obj)   {
   if (obj == null)  return false;
   return obj.Amount == this.Amount && obj.Unit == this.Unit;
  }
}
class Program  {
   static void Main(string[] args)   {
     Money m1 = new Money(100, "RUR");
```

Перевантаження методів Equals, GetHashCode. Equals та «==»

Метод GetHashCode

Даний метод повертає **хеш-код** - число відповідне значення об'єкта.

- для одного і того ж об'єкта повинен бути однаковий хеш-код

- для двох рівних об'єктів хеш-код повинен бути однаковим. Тільки це не означає, що якщо об'єкти нерівні, то їх хеш-коди обов'язково будуть різними.

Методи *Equals* і *GetHashCode* тісно пов'язані між собою, при перевизначенні одного з них, слід ігнорувати і іншого. Базова реалізація методу *GetHashCode* в класі *object* дуже умовна, і вона не забезпечує друге властивість, коли однакові об'єкти мають однакові хеш-коди.

```
public class Money {
   public decimal Amount { get; set; }
   public string Unit { get; set; }
   public Money(decimal amount, string unit) {
     Amount = amount; Unit = unit; }
   public override bool Equals(object obj) {
     if (obj == null)  return false;
     Money m = obj as Money;
     if (m as Money == null)  return false;
     return m.Amount == this.Amount && m.Unit == this.Unit;
   }
public override int GetHashCode()  {
     int unitCode;
     if (Unit == "RUR")
       unitCode = 1;
```

Регулярні вирази. Клас Regex

Регулярний вираз – це якийсь шаблон, складений із символів і спецсимволов, який дозволяє знаходити відповідні підрядки цим шаблоном в інших рядках. Спецсимволов і різних правил їх комбінування є дуже багато, тому регулярні вирази можна навіть назвати таким собі окремою мовою програмування. Ті, хто користувався пошуком файлів в Windows можуть знати, що для того, щоб знайти файли тільки заданого розширення, задається шаблон типу «*.txt». Тут «*» - спецсимвол, який означає будь-які імена файлів. Так ось регулярні вирази надають подібний механізм.

Регулярні вирази. Клас Regex

Регулярні вирази надають масу можливостей, деякі з них:

- замінити в рядку всі однакові слова іншим словом, або видаляти такі слова;
- виділяти з рядка необхідну частину. Наприклад, з будь посилання (http://mycsharp.ru/post/33/2013_10_19_virtualnye_metody_v_si-sharp_pereopredelenie_metodov.html) выделять только доменную часть (mycsharp.ru);
- перевіряти відповідає рядок заданому шаблону. Наприклад, перевіряти, чи правильно введено email, телефон тощо;
- перевіряти, чи містить рядок задану підрядок;
- витягувати з рядка всі входження підрядків, що відповідають шаблону регулярного виразу. Наприклад, отримати всі дати з рядка.

Регулярні вирази. Клас Regex

Для того, щоб працювати з регулярними виразами необхідно підключити на початку програми простір імен *using System.Text.RegularExpressions;*

В Сі-шарп роботу з регулярними виразами надає клас *Regex*.

Створення регулярного виразу має наступний вигляд:

```
Regex myReg = new Regex([шаблон]);
```

Тут [шаблон] – це рядок, що містить символи та спеціальні символи.

У *Regex* також є і другий конструктор, який приймає додатковий параметр – опції пошуку. Це ми розглянемо далі.

Регулярні вирази. Клас Regex

```
static void Main(string[] args)
{
   string data1 = "Петр, Андрей, Николай";
   string data2 = "Петр, Андрей, Александр";
   Regex myReg = new Regex("Николай"); // створення регулярного виразу
   Console.WriteLine(myReg.IsMatch(data1)); // True
   Console.WriteLine(myReg.IsMatch(data2)); // False
   Console.ReadKey();
}
```

Регулярні вирази. Клас Regex

IsMatch – перевіряє містить рядок хоча б одну підрядок відповідну шаблоном регулярного виразу. Робота цього методу показано в прикладі вище.

Match – повертає першу підрядок, що відповідає шаблону, у вигляді об'єкта класу Match.

Клас Match надає різну інформацію про підрядку – довжину, індекс, значення та інше.

```
string data1 = "Петр, Андрей, Николай";    Regex myReg = new Regex("Николай");
Match match = myReg.Match(data1);
Console.WriteLine(match.Value); // "Николай"
Console.WriteLine(match.Index); // 14
Console.ReadKey();
```

Регулярні вирази. Клас Regex

Matches – повертає всі відповідні підрядки шаблоном у вигляді колекції типу *MatchCollection*. Кожен елемент цієї колекції типу *Match*.

```
static void Main(string[] args)  {
   string data1 = "Петр, Николай, Андрей, Николай";
   Regex myReg = new Regex("Николай");
   MatchCollection matches = myReg.Matches(data1);
   Console.WriteLine(matches.Count); // 2
   foreach (Match m in matches)
      Console.WriteLine(m.Value); //висновок всіх підрядків "Николай"
   Console.ReadKey();
}
```

Регулярні вирази. Клас Regex

Replace – повертає рядок, в якій замінені усі підрядки, що відповідають шаблону, новим рядком:

```
static void Main(string[] args)
{
   string data1 = "Петр, Николай, Андрей, Николай";
   Regex myReg = new Regex("Николай");
   data1 = myReg.Replace(data1, "Максим");
   Console.WriteLine(data1); //"Петр, Максим, Андрей, Максим"
   Console.ReadKey();
}
```

Регулярні вирази. Клас Regex

Split - повертає масив рядків, отриманий в результаті поділу входить рядка в місцях відповідності шаблону регулярного виразу:

```
static void Main(string[] args)
{
  string data1 = "Петр,Николай,Андрей,Николай";
  Regex myReg = new Regex(",");
  string[] names = myReg.Split(data1); // масив імен
  Console.ReadKey();
}
```

Регулярні вирази. Клас Regex

Класи символів

Позначення	Опис	Шаблон	Відповідність
[група символів]	Будь-який з перелічених у дужках символів. Використовуючи тирі можна вказати діапазон символів, наприклад, [a-f] - те ж саме, що [abcdef]	[abc]	«a» в «and»
[^група символів]	Будь-який символ, крім перелічених у дужках	[^abc]	«n», «d» в «and»
\d	Цифра. Еквівалентно [0-9]	\d	«1» в «data1»
\D	Будь-який символ, крім цифр. Еквівалентно [^0-9]	\D	«y» в «2014y»

Позначення	Опис	Шаблон	Відповідність
\w	Цифра, буква (латинський алфавіт) або знак підкреслення. Еквівалентно [0-9a-zA-Z_]	\w	«1», «5», «c» в «1.5c»
\W	Будь-який символ, крім цифр, букв (латинський алфавіт) і знака підкреслення. Еквівалентно [^0-9a-zA-Z_]	\W	«.» в «1.5c»
\s	Пробільний символ (пробіл, табуляція, переклад рядка і т. п.)	\s	« » в «c sharp»
\S	Будь-який символ, крім пробільних	\S	«r» «p» в «c sharp»
.	Будь-який символ, крім переведення рядка. Для пошуку будь-якого символу, включаючи переклад рядка, можна використовувати конструкцію [\s\S]	c.harp	«csharp» в «mycsharp»

Позначення	Опис (**Символи повторення**)	Шаблон	Відповідність
*	Відповідає попереднього елемента нуль або більше раз	\d*.	«a», «1b»в «a1b23c»
+	Відповідає попереднього елемента один або більше раз	\d+.	«1b», «23c» в «a1b23c»
?	Відповідає попереднього елемента нуль або один раз	\d?\D	«a», «1b», «a1b23c»
{n}	Відповідає попереднього елемента, який повторюється рівно n разів	\d{1}	«43» в «2,43,5,82»
{n,}	Відповідає попереднього елемента, який повторюється щонайменше n разів	\d{1,}	«43» в «2,43,5462»
{n,m}	Відповідає попереднього елемента, який повторюється щонайменше n разів і максимум m	\d{1,3}	«43» в «2,43,546»

Символи прив'язки

Позначення	Опис	Шаблон	Відповідність
^	Відповідність повинна знаходитися на початку рядка	^\d{2}	«32» в «32,43,54»
$	Відповідність повинна знаходитися в кінці рядка або до символу \n при багаторядковому пошуку	\d{2}$	«54» в «32,43,54»
\b	Відповідність повинна знаходитися на кордоні алфавітно-цифрового символу (\w) і не алфавітно-цифрового (\W)	\b\d{2}	«32», «54» в «32 a43 54»
\B	Відповідність не повинно знаходитися на кордоні	\B\d{2}	«43» в «32 a43 54»
\G	Відповідність повинна перебувати на позиції кінця попереднього відповідності	\G\d	«3», «2», «4» в «324.758»

Символи вибору

Позначення	Опис	Шаблон	Відповідність
|	Працює як логічне «АБО» - відповідає першого та/або другого шаблону	one|two	«one», «two» в «one two three»
(группа символов)	Групує набір символів в єдине ціле для якого далі можуть використовуватися * ? і т. д. Кожній такій групі призначається порядковий номер зліва направо починаючи з 1. За цим номером можна посилатися на групу \номер_групи	(one)\1	«oneone» в «oneone onetwoone»
(?:группа символов)	Та ж угрупування тільки без призначення номера групи	(?:one){2}	«oneone» в «oneone onetwoone»

Інші символи

Позначення	Опис	Шаблон	Відповідність
\t	Символ табуляції	\t	
\v	Символ вертикальної табуляції	\v	
\r	Символ повернення каретки	\r	
\n	Символ переведення рядка	\n	
\f	Символ переведення сторінки	\f	
\	Символ, який дозволяє додавати спеціальні символи, щоб ті сприймалися буквально. Наприклад, щоб була відповідність символу зірочки, шаблон буде виглядати так*	\d\.\d	«1.1», «1.2» в «1.1 1.2»

Регулярні вирази. Клас Regex

```
static void Main(string[] args){
   Regex myReg = new Regex(@"[A-Za-z]+[\.A-Za-z0-9_-]*[A-Za-z0-9]+@[A-Za-z]+\.[A-Za-z]+");
   Console.WriteLine(myReg.IsMatch("email@email.com")); // True
   Console.WriteLine(myReg.IsMatch("email@email")); // False
   Console.WriteLine(myReg.IsMatch("@email.com")); // False
   Console.ReadKey();
}
```

Тут перед початком рядка регулярного виразу стоїть символ «@» який вказує комплятору сприймати всі символи буквально. Це необхідно, щоб коректно сприймався символ «\».

Регулярні вирази. Клас Regex

Тут ми поговоримо про другий конструкторі Regex, що приймає в якості другого аргументу значення перерахування RegexOptions. У цьому перерахування є наступні значення:

IgnoreCase – ігнорування регістру при пошуку. Знаходить відповідності незалежно великими або малими літерами в рядку написано слово;

RightToLeft – пошук буде виконаний справа наліво, а не зліва направо;

Multiline – багаторядковий режим пошуку. Змінює роботу спецсимволов «^» і «$» так, що вони відповідають початку і кінця кожного рядка, а не тільки початку і кінця цілої рядка;

Singleline – однорядковий режим пошуку;

Регулярні вирази. Клас Regex

CultureInvariant - ігнорування національних установок рядка;

ExplicitCapture – забезпечується пошук тільки буквальних збігів;

Compiled – регулярний вираз компілюється в збірку, що робить більш швидким його виконання але збільшує час запуску;

IgnorePatternWhitespace – ігнорує в шаблоні усі неекрановані прогалини. З цим параметром шаблон «a b буде аналогічним шаблоном «ab»;

None – використовувати пошук за замовчуванням.

Регулярні вирази. Клас Regex

Приклад програми з використанням параметра пошуку (ігнорування регістра):

```
string data = "nikolay, sergey, oleg";
Regex myRegIgnoreCase = new Regex(@"Sergey", RegexOptions.IgnoreCase);
Regex myReg = new Regex(@"Sergey");
Console.WriteLine(myRegIgnoreCase.IsMatch(data)); // True
Console.WriteLine(myReg.IsMatch(data)); // False
Console.ReadKey();
```

Якщо необхідно встановити декілька параметрів, тоді вони поділяються оператором порозрядного «АБО» - «|»

```
Regex myReg = new Regex(@"Sergey", RegexOptions.IgnoreCase |
RegexOptions.IgnorePatternWhitespace);
```

Регулярні вирази. Клас Regex

```
public static string GetDomain(string url)  {
   Regex re = new Regex("http://", RegexOptions.IgnoreCase);
   url = re.Replace(url, "");
   Regex reWww = new Regex(@"www\.", RegexOptions.IgnoreCase);
   url = reWww.Replace(url, "");
   int end = url.IndexOf("/");
   if (end != -1)    url = url.Substring(0, end);
   return url;
}
static void Main(string[] args) {
   string url1 = "http://mycsharp.ru/post/33/2013_10_19_.html";
```

Форматування рядків

В Сі-шарп можливістю задати форматування мають наступні методи:

- System.String.Format
- Console.WriteLine
- StreamWriter.Write
- ToString

Методи *WriteLine* і *Write* використовуються для виведення інформації в консоль, і при цьому дають можливість відформатувати висновок.

Метод *Format* класу *String* призначений конкретно для форматування. Він повертає відформатовану рядок. Різниці між самим форматуванням для цих методів немає. Форматування в методі *ToString* можна вказати тільки для чисел і дат.

Форматування рядків

Загальна структура форматування рядків має наступний вигляд:

String.Format("рядок формату", arg0, arg1, …, argn);

arg0 и arg1 тут – аргументи форматування (числа, рядки, дати і т. д.), з яких в результаті буде створена нова відформатована рядок.

Рядок формату може містити звичайні символи, які будуть відображені у тому вигляді, в якому вони задані, і команди форматування. Команда форматування полягає у фігурні дужки і має наступну структуру:

{[номер аргументу], [ширина]:[формат]}

Форматування рядків

За [номером аргументу] вказується до якого аргументу буде застосована дана команда (відлік аргументів починається з нуля).

[ширина] задає мінімальний розмір поля. [формат] – специфікатор формату.

Параметри [ширина] і [формат] не є обов'язковими. Приклад простого форматування:

```
string formattedString = string.Format("Result is {0}", 5); // "Result is 5"
```

Тут на місце команди {0} підставляється 0-й аргумент.

```
int num1 = 5, num2 = 3;
string formattedString = string.Format("{0}+{1}={2}", num1, num2, num1+num2); // "5+3=8"
Console.WriteLine(formattedString);  Console.ReadLine();
```

Форматування рядків

Параметр "ширина"

Іноді необхідно відформатувати

рядки, що містять числа, щоб вони

були вирівняні по лівому або правому

краю і мали однакову довжину.

До числа, яке має менше

символів, ніж значення ширини, будуть додано пробіли зліва

(позитивна ширина) або праворуч

(негативна ширина):

```
Console.WriteLine("Result is {0, 6}", 1.2789);
Console.WriteLine("Result is {0, 6}", 7.54);
```

Форматування рядків

Вбудовані формати числових даних

А тепер ми розглянемо параметр команди форматування після двокрапки – формат.

Спеціальний символ	Формат
c/C	Грошова одиниця
d/D	Цілі числа
e/E	Експоненціальні числа
f/F	Числа з фіксованою точкою

Форматування рядків

Спеціальний символ	Формат
n/N	Числа з фіксованою крапкою з відділенням груп розрядів
p/P	Відсотки
r/R	Формат кругового перетворення. Тільки фіксована точка
x/X	Шістнадцяткові числа

```
Console.WriteLine("{0:c}", 5.50); // "5,50 грн."

Console.WriteLine("{0:c1}", 5.50); // "5,5 грн."

Console.WriteLine("{0:e}", 5.50); // "5,500000e+000"

Console.WriteLine("{0:d}", 32); // "32"

Console.WriteLine("{0:d4}", 32); // "0032"

Console.WriteLine("{0:p}", 0.55); // "55,00%"
```

Форматування рядків

Користувальницький формат числових даних

Спеціальний символ	Значення
0	Цифра або нуль
#	Цифра
.	Роздільник дробу
,	Роздільник тисяч
%	Відсоток
e	Експонента
;	Визначає розділи, які описують формати для додатних, від'ємних чисел і нуля
\	Екранування спеціальних символів. Дозволяє вставляти спец-символи як текст

Форматування рядків

Приклад використання користувальницьких форматів:

```
Console.WriteLine("{0:0000.00}", 1024.32); // "1024,32"
Console.WriteLine("{0:00000.000}", 1024.32); // "01024,320"
Console.WriteLine("{0:####.###}", 1024.32); // "1024,32"
Console.WriteLine("{0:####.#}", 1024.32); // "1024,3"
Console.WriteLine("{0:#,###.##}", 1024.32); // "1 024,32"
Console.WriteLine("{0:##%}", 0.32); // "32%"
Console.WriteLine("{0:<####.###>;[####.###];нуль}", 1024.32); // "<1024,32>"
Console.WriteLine("{0:<####.###>;[####.###];нуль}", -1024.32); // "[1024,32]"
Console.WriteLine("{0:<####.###>;[####.###];нуль}", 0); // "нуль"
```

Тут варто відзначити, що якщо кількість цифр цілої частини числа більше ніж кількість символів «0» або «#» у форматі, ціла частина числа все одно буде виводитися повністю.

Форматування рядків

Вбудовані формати дати і часу

Для роботи з датою і часом існує окремий набір стандартних форматів. Дані формати наведені в таблиці:

Спеціальний символ	Формат	Приклад
d	Коротка дата	30.06.2014
D	Довга дата	30 червня 2014 р.
t	Короткий час	22:30
T	Довгий час	22:30:10
f	Довга дата/короткий час	30 червня 2014 р. 22:30
F	Довга дата/довгий час	30 червня 2014 р. 22:30:10

Форматування рядків

g	Коротка дата/короткий час	30.06.2014 22:30
G	Коротка дата/довгий час	30.06.2014 22:30:10
M/m	Місяць і день	июня 30
O/o	Зворотний	2014-06-30T22:30:10.0000000
R/r	RFC1123	Mon, 30 Jun 2014 22:30:10 GMT
s	Для сортування	2014-06-30T22:30:10
u	Локальне, в універсальному форматі	2014-06-30 22:30:10Z
U	GMT	30 июня 2014 г. 19:30:10
Y	Рік і місяць	Июнь 2014

Форматування рядків

Користувальницький формат дати і часу (на прикладі дати 30.06.2014 22:30:10.1234):

Спеціальний символ	Значення	Результат
y	Рік 0-9	4
yy	Рік 00-99	14
yyyy	Рік, 4 цифри	2014
M	Місяць 1-12	6
MM	Місяць 01-12	06
d	День 1-31	0
dd	День 01-31	30
h	Годину 1-12	2

Форматування рядків

Спеціальний символ	Значення	Результат
hh	Годину 01-12	10
H	Годину 1-24	22
H	Годину 01-24	22
m	Хвилина 0-59	30
mm	Хвилина 00-59	30
s	Секунда 0-59	10
ss	Секунда 00-59	10
f – ffffff	Частки секунди	12 (для ff)
F-FFFFFF	Частки секунди, якщо вони не дорівнюють 0	12 (для ff)
MMM	Скорочене ім'я місяця	Чер
MMMM	Ім'я місяця	Червень
ddd	Скорочене ім'я дні тижня	Пн
dddd	Ім'я дні тижня	понеділок

Форматування рядків

Спеціальний символ	Значення	Результат
tt	Маркер для "AM" і "PM" 12-годинного формату	PM
zz	Зміщення тимчасової зони, короткий	+03
zzz	Зміщення тимчасової зони, повне	+03:00
gg	Ера	A.D.
:	Роздільник часу	22:30:10
/	Роздільник дати	30.06.2014
\	Екранування	

Приклад використання стандартних форматів дати і часу:

```
Console.WriteLine("{0:d}", DateTime.Now); // "30.06.2014"

Console.WriteLine("{0:D}", DateTime.Now); // "30 червня 2014 р."

Console.WriteLine("{0:t}", DateTime.Now); // "2:57"

Console.WriteLine("{0:T}", DateTime.Now); // "2:57:53"

Console.WriteLine("{0:U}", DateTime.Now); // "29 червня 2014 р. 23:57:53"

Console.WriteLine("{0:Y}", DateTime.Now); // "Червень 2014 р."
```

Приклад використання користувальницьких форматів дати і часу:

```
Console.WriteLine("{0:y yy yyyy yyyy}", DateTime.Now); // "4 14 2014 2014"

Console.WriteLine("{0:d dd ddd dddd}", DateTime.Now); // "30 30 Пн понеділок"

Console.WriteLine("{0:M MM MMM}", DateTime.Now); // "6 06 Чер"

Console.WriteLine("{0:HH.mm.ss dd-MMM-yyyy}", DateTime.Now); // "03.21.22 30-Чер-2014"

Console.WriteLine("{0:z zz zzz}", DateTime.Now); // "+3 +03 +03:00"
```

Форматування рядків

Регіональні параметри CultureInfo

У прикладах вище стандартні параметри культури, щоб змінити ці параметри є *CultureInfo*. За замовчуванням *CultureInfo* відповідає налаштувань Windows. Зміна цих параметрів показано прикладами

```
string formattedString = string.Format(new System.Globalization.CultureInfo("en-US"), "{0:dddd} Money - {1:c}", DateTime.Now, 15);
Console.WriteLine("{0:dddd} Money - {1:c}", DateTime.Now, 15);
// "понеділок Money - 15,00 руб."
Console.WriteLine(formattedString); // "Monday Money - $15.00"
formatedString = string.Format(new System.Globalization.CultureInfo("uk-UA"), "{0:dddd} Money - {1:c}", DateTime.Now, 15);
Console.WriteLine(formattedString); // "понеділок Money - 15,00 грн."
```

Делегати

Крім властивостей і методів класи можуть містити делегати та події.

Делегати представляють такі об'єкти, які вказують на інші методи.

Тобто делегати - це покажчики методи. З допомогою делегатів ми можемо викликати певні методи у відповідь на відбулися деякі дії.

Тобто, по суті, делегати розкривають нам функціонал функцій зворотного виклику.

```
delegate int Operation(int x, int y);

delegate void GetMessage();
```

Делегати

```
class Program  {
    delegate void GetMessage(); // 1. Оголошуємо делегат
    static void Main(string[] args)  {
        GetMessage del; // 2. Створюємо змінну делегата
        if (DateTime.Now.Hour < 12)  {
            del = GoodMorning; // 3. Присвоюємо змінної адреса методу
        }  else  {
            del = GoodEvening;
        }
        del.Invoke(); // 4. Викликаємо метод          Console.ReadLine();
    }
    private static void GoodMorning()  {
```

Делегати

```
class Program {
    delegate int Operation(int x, int y);
    static void Main(string[] args)  {
        // присвоювання адреси методу через контруктор
        Operation del = new Operation(Add);
// делегат вказує на метод Add
        int result = del.Invoke(4,5);
        Console.WriteLine(result);
        del = Multiply;
// тепер делегат вказує на метод Multiply       result = del.Invoke(4, 5);
        Console.WriteLine(result);
        Console.Read();
    }
```

Делегати

```
class Program
{
    delegate void GetMessage();
    static void Main(string[] args) {
        if (DateTime.Now.Hour < 12)  {
            Show_Message(GoodMorning);
        }  else  {
             Show_Message(GoodEvening);
        }
        Console.ReadLine();
    }
    private static void Show_Message ( GetMessage _del) {
```

Делегати

Дані приклади, можливо, не показують істинної сили делегатів, так як потрібні нам методи в даному випадку ми можемо викликати і безпосередньо без всяких делегатів. Однак найбільш сильна сторона делегатів полягає в тому, що вони дозволяють створити функціонал методів зворотного виклику, повідомляючи інші об'єкти про минулі події.

Делегати

```
class Account {
    int _sum; // Змінна для зберігання суми
    int _percentage; // Змінна для зберігання відсотка
    public Account(int sum, int percentage)  {
        _sum = sum;  _percentage = percentage;
    }
    public int CurrentSum  {
        get { return _sum; }
    }
    public void Put(int sum)  {
        _sum += sum;
    }
    public void Withdraw(int sum)  {
```

Делегати

Припустимо, у разі виведення грошей з допомогою методу Withdraw нам треба якось повідомляти про це самого клієнта і, можливо, інші об'єкти. Для цього створимо делегат AccountStateHandler.

```
class Account  {
public delegate void AccountStateHandler(string message);      // Оголошуємо делегат
AccountStateHandler del;        // Створюємо змінну делегата
   // Реєструємо делегат
   public void RegisterHandler(AccountStateHandler _del)  {
      del = _del;
   }     // Далі інші рядки класу Account
```

Делегати

```
public void Withdraw(int sum) {
    if (sum <= _sum) {
        _sum -= sum;
        if (del != null)  del("Сумма " + sum.ToString() + " знята з рахунку");
    }  else  {
        if (del != null)  del("Недостатньо грошей на рахунку");
} }
```

Тепер при знятті коштів через метод Withdraw ми спочатку перевіряємо, чи має делегат посилання на який-небудь метод (інакше він має значення null). І якщо метод встановлений, то викликаємо його, передаючи відповідне повідомлення в якості параметра.

Делегати

```
class Program  {
    static void Main(string[] args)   {
        Account account = new Account(200, 6); // створюємо банківський рахунок
        // Додаємо у делегат посилання на метод Show_Message
        // а сам делегат передається як параметр методу RegisterHandler
        account.RegisterHandler(new Account.AccountStateHandler(Show_Message));
        account.Withdraw(100);  // Два рази поспіль намагаємося зняти гроші
        account.Withdraw(150);
        Console.ReadLine();
    }
    private static void Show_Message(String message)  {   Console.WriteLine(message);
} }
```

Делегати

```
Сумма 100 снята со счета
Недостаточно денег на счете
```

Таким чином, ми створили механізм зворотного виклику для класу Account, який спрацьовує у разі зняття грошей. Оскільки делегат оголошено всередині класу Account, то щоб до нього доступ, використовується вираз Account.AccountStateHandler. Знову ж може виникнути питання: чому б до коді методу Withdraw() не виводити повідомлення про зняття грошей? Навіщо потрібно задіяти якийсь делегат? Справа в тому, що не завжди у нас є доступ до коду класів. Наприклад, частина класів може створюватися і компілюватися однією людиною, яка не буде знати, як ці класи будуть використовуватися. А використовувати ці класи буде інший розробник. Так, тут ми виводимо повідомлення на консоль. Однак для класу Account не важливо, як це повідомлення виводиться. Класу Account навіть не відомо, що взагалі буде робитися в результаті списання грошей. Він просто надсилає повідомлення про це через делегат.

Події

В минулій темі ми розглянули, як з допомогою делегатів можна створювати механізм зворотних викликів у програмі. Однак C# для тієї ж мети надає більш зручні і прості конструкції під назвою події, які сигналізують системі про те, що сталося певне дію.

Події оголошуються в класі з допомогою ключового слова event, після якого йде назва делегата:

```
public delegate void AccountStateHandler(string message); // Оголошуємо делегат
public event AccountStateHandler Withdrowed; //  Подія, що виникає при виведенні грошей
```

Зв'язок з делегатом означає, що метод, який обробляє дану подію, повинен приймати ті ж параметри, що і делегат, і повертати той же тип, що і делегат.

```
class Account  {
    // Оголошуємо делегат
    public delegate void AccountStateHandler(string message);
    // Подія, що виникає при виведенні грошей
    public event AccountStateHandler Withdrowed;
    // Подія, що виникає при додавання на рахунок
    public event AccountStateHandler Added;
    int _sum; // Змінна для зберігання суми
    int _percentage; // Змінна для зберігання відсотка
    public Account(int sum, int percentage)  {
        _sum = sum;
        _percentage = percentage;
    }
    public int CurrentSum  {
        get { return _sum; }
    }
    public void Put(int sum)   {
        _sum += sum;
        if (Added != null)
            Added("На рахунок надійшло " + sum);
```

Події

Тут ми визначили дві події: Withdrowed і Added. Обидві події оголошені як екземпляри делегата AccountStateHandler, тому для обробки цих подій потрібна метод, що приймає рядок у якості параметра.

Потім в методах Put і Withdraw ми викликаємо ці події. Перед викликом ми перевіряємо, закріплені за цими подіями обробники (if (Withdrowed != null)). Так як ці події представляють делегат AccountStateHandler, що приймає в якості параметра рядок, то і при виклику подій ми передаємо в них рядок

Події

```
class Program
{
    static void Main(string[] args)
    {
        Account account = new Account(200, 6);
        // Додаємо обробник події
        account.Added += Show_Message;
        account.Withdrowed += Show_Message;
        account.Withdraw(100);
        // Видаляємо обробник події
        account.Withdrowed -= Show_Message;
        account.Withdraw(50);
        account.Put(150);

        Console.ReadLine();
```

Події

Для прикріплення обробника події до певної події використовується операція += і відповідно для відкріплення – операція -=: подія += метод_обробника_події. Знову ж таки звертаю увагу, що метод обробника повинен мати такі ж параметри, як і делегат події, і повертати той же тип. В результаті ми отримаємо наступний ко

```
Сумма 100 снята со счета
На счет поступило 150
```

Крім використаного вище способи прикріплення обробників є й інший з використанням делегата. Але обидва способи будуть рівноцінні:

```
account.Added += Show_Message;
account.Added += new Account.AccountStateHandler(Show_Message);
```

Події

Клас події даних AccountEventArgs

Якщо раптом ви коли-небудь створювали графічні додатки Windows Forms або WPF, то, ймовірно, стикалися з обробниками, які в якості параметра приймають аргумент типу EventArgs, наприклад, оброблювач натискання кнопки private void button1_Click(object sender, System.EventArgs e){} Параметр e, будучи об'єктом класу EventArgs, містить всі дані події. Додамо і в нашу програму подібний клас. Назвемо його AccountEventArgs і додамо в нього наступний код:

Події

```
class AccountEventArgs
{
    // Повідомлення
    public string message;
    // Сума, на яку змінився рахунок
    public int sum;
    public AccountEventArgs(string _mes, int _sum)
    {
        message = _mes;
        sum = _sum;
    }
}
```

```
class Account {
    // Оголошуємо делегат
    public delegate void AccountStateHandler(object sender, AccountEventArgs e);
    // Подія, що виникає при виведенні грошей
    public event AccountStateHandler Withdrowed;
    // Подія, що виникає при додаванні на рахунок
    public event AccountStateHandler Added;
    int _sum; // Змінна для зберігання суми
    int _percentage; // Змінна для зберігання відсотка
   public Account(int sum, int percentage)  {
        _sum = sum; _percentage = percentage;
    }
    public int CurrentSum  {   get { return _sum; }  }
    public void Put(int sum)  {
        _sum += sum;
        if (Added != null)  Added(this, new AccountEventArgs("На счет поступило " + sum, sum));
    }
```

Події

```
class Program  {
    static void Main(string[] args) {
        Account account = new Account(200, 6);
        // Додаємо обробник події
        account.Added += Show_Message;
        account.Withdrowed += Show_Message;
        account.Withdraw(100);
        // Видаляємо обробник події
        account.Withdrowed -= Show_Message;
        account.Withdraw(50);
        account.Put(150);
```

Анонімні методи

Іноді такі методи потрібні для обробки однієї події і більше цінності не представляють і ніде не використовуються. Анонімні методи дозволяють вбудувати код там, де він викликається, наприклад:

```
Account account = new Account(200, 6);
// Додаємо обробник події
account.Added += delegate(object sender, AccountEventArgs e)  {
    Console.WriteLine("Сума транзакції: {0}", e.sum);
    Console.WriteLine(e.message);
};
```

Анонімні методи

І важливо відзначити, що на відміну від блоку методів або умовних і циклічних конструкцій, блок анонімних методів повинен закінчуватися крапкою з комою після закриває фігурної дужки. Якщо для анонімного методу не потрібно параметрів, то він використовується без дужок:

```
delegate void GetMessage();
static void Main(string[] args)   {
    GetMessage message = delegate {
        Console.WriteLine("анонімний делегат");
    };
    message();
    Console.Read();
}
```

Лямбды

Лямбда-вирази являють спрощену запис анонімних методів. Лямбда-вирази дозволяють створити ємні лаконічні методи, які можуть повертати деяке значення і які можна передати в якості параметрів в інші методи.

Ламбда-вирази мають наступний синтаксис: зліва від лямбда-оператора => определяется список параметрів, а праворуч блок виразів, що використовує ці параметри: (список_параметрів) => выражение. Например:

```
class Program {
    delegate int Square(int x); // оголошуємо делегат, що приймає int і повертає int
    static void Main(string[] args) {
        Square squareInt = i => i * i; // об'єкту делегата присвоюється лямбда-вираз
        int z = squareInt(6); // використовуємо делегат
        Console.WriteLine(z); // виводить число 36
        Console.Read();
} }
```

Лямбди

Тут i => i * i являє лямбда-вираз, де i - це параметр, а i*i - вираз.

При використанні треба враховувати, що кожний параметр в лямбда-виразу неявно перетворюється у відповідний параметр делегата, тому типи параметрів повинні бути однаковими.

Крім того, кількість параметрів повинно бути таким же, як і у делегати. І обчислене значення лямбда-виразів має бути тим же, що і у делегата.

Візьмемо клас Account з минулої теми і перепишемо прикріплення обробника події за допомогою лямбды-вираз:

```
Account account = new Account(200, 6);
account.Added += (sender, e)=> {
    Console.WriteLine("Сумма транзакции: {0}", e.sum);
    Console.WriteLine(e.message);
};
```

Лямбди

Оскільки тут використовується кілька параметрів, то вони беруться в дужки. І так як в тілі лямбда-вирази застосовується кілька виразів, то вони полягають у блок з фігурних дужок.

Буває, що не потрібно. У цьому випадку замість параметра в лямбда-виразу використовуються порожні дужки:

```
class Program  {
    delegate void message(); // делегат без параметрів
    static void Main(string[] args)  {
        message GetMessage = () => { Console.WriteLine("Лямбда-вираз"); };
        GetMessage();
        Console.Read();
}  }
```

Лямбди

Також лямбда-вираз необов'язково має приймати блок операторів та виразів. Воно може також приймати посилання на метод:

```
class Program {
    delegate void message(); // делегат без параметрів
    static void Main(string[] args)  {
        message GetMessage = () => Show_Message();
        GetMessage();
    }
    private static void Show_Message() {
        Console.WriteLine("Привіт світ!");
    }
}
```

Лямбди

```
class Program   {
    delegate bool IsEqual(int x);
    static void Main(string[] args) {
        int[] integers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        // знайдемо суму чисел більше 5
        int result1 = Sum(integers, x => x > 5);
        Console.WriteLine(result1); // 30
        // знайдемо суму парних чисел
int result2 = Sum(integers, x => x % 2 == 0);
        Console.WriteLine(result2);  //20
        Console.Read();
    }
    private static int Sum (int[] numbers, IsEqual func)
```

Лямбди

Метод Sum приймає в якості параметра масив чисел і делегат IsEqual і повертає суму чисел масиву у вигляді об'єкта int. У циклі проходимо по всіх чисел і складаємо їх. Причому складаємо тільки ті числа, для яких делегат IsEqual func повертає true. Тобто делегат IsEqual тут фактично визначає критерії, яким повинні відповідати значення масиву. Але на момент написання методу Sum нам невідомо, що це за умову. При виклику методу Sum йому передається масив і лямбда-вираз:

```
int result1 = Sum(integers, x => x > 5);
```

Лямбди

Тобто параметр x тут буде представляти число, яке передається у делегат:

```
if (func(i))
```

А вираз x > 5 являє умова, якій має відповідати число. Якщо число відповідає цій умові, то лямбда-вираз повертає true, а передане число складається з іншими числами. Подібним чином працює другий виклик методу Sum, тільки тут вже йде перевірка на парність числа, тобто якщо залишок від ділення на 2 дорівнює нулю:

```
int result2 = Sum(integers, x => x % 2 == 0);
```

Продовження може буде!!!