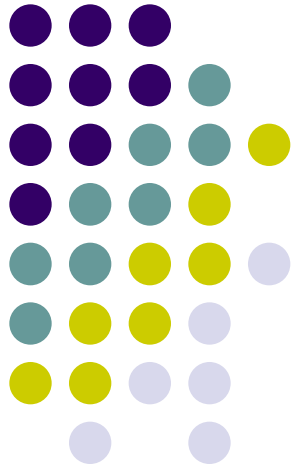
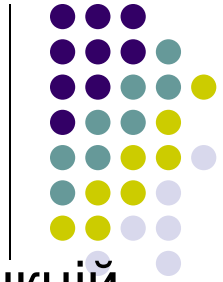
Класи і структури C#

константи, поля, події, методи,
властивості, конструктори, фіналізатори,
операції, індексатори.
методи, що розширяють клас.
конструювання, копіювання об’єктів.
наслідування, використання інтерфейсів



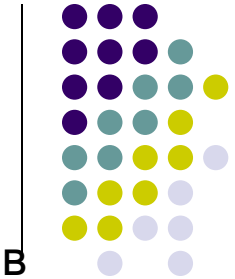
class VS struct



- сукупність даних і функцій
- тип-посилання (купа)
- наслідують System.Object
- підтримують одинарне наслідування класів, множинне – інтерфейсів
- конструктор за замовчанням генерує компілятор; можна перевизначити; оголошення інших конструкторів скасовує автоматичну генерацію
- поля можна ініціалізувати в оголошенні класу

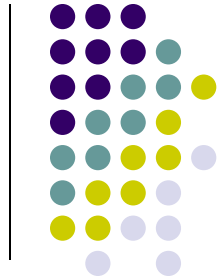
- сукупність даних і функцій
- тип-значення (стек)
- наслідують System.ValueType
- підтримують множинне наслідування інтерфейсів, не підтримують – структур
- конструктор за замовчанням компілятор генерує завжди; його не можна перевизначити
- ініціалізувати поля в оголошенні структури заборонено

Структура класу



- Дані-члени класу:
  - константи (неявно статичні) – для потреб класу чи його клієнтів
  - поля класу (статичні поля) – спільні для всіх екземплярів
  - поля екземпляра – стан окремого об’єкта
  - події класу, події екземпляра – це засіб повідомлення про щось варте уваги
- Функції-члени класу:
  - методи класу – доступ до полів класу, поведінка класу
  - методи екземпляра – маніпулюють полями екземпляра
  - властивості – набори функцій, доступ до яких нагадує доступ до поля
  - конструктори – функції без типу з іменем класу для ініціалізації екземплярів
  - фіналізатори – ім'я класу з тильдою, працюють перед знищенням екземпляра
  - операції – оператори (C++), перевантажують відомі операції для типів користувача
  - індексатори – індексують об'єкт як масив чи колекцію

Оголошення класу



```
[модифікатор доступу] class ім’я_класу
{
// дані: константи, поля, події
    [модифікатори] тип ім’я [ініціалізатор];
    ...
// функції: методи, властивості, операції ...
    [модифікатори] тип ім’я_методу ([параметри])
    {
     тіло методу
    }
    ...
}
```

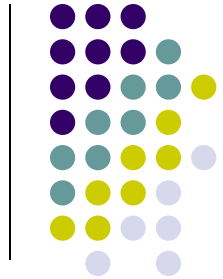
Модифікатори доступу C#

модифікатор	до чого застосовують	призначення
public	тип, член типу	відкриті елементи не мають обмежень доступу
private	член типу, вкладений тип	закриті елементи доступні тільки в класі (структурі)
protected	член типу, вкладений тип	захищені елементи доступні тільки класові та його підкласам
internal	тип, член типу	внутрішні елементи доступні в межах assembly
protected internal	член типу, вкладений тип	елемент доступний всередині assembly та класу (підкласів)

Інші модифікатори C#

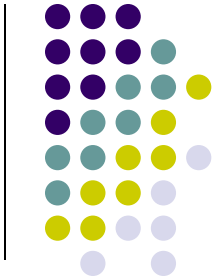
модифікатор	до чого застосовують	призначення
static	клас, поле, метод, властивість	описують дані, поведінку класу
abstract	метод, клас	задати протокол взаємодії, не створювати екземпляри
virtual	метод, властивість	реалізувати пізнє зв'язування, поліморфізм поведінки
override	метод, властивість	
new	метод	перекрити батьківський метод
sealed	клас, метод	заборонити наслідування, перевизначення
extern	статичний метод	методи написані іншою мовою
const	поле	оголосити в класі константу
readonly	(відкрите) поле	гнучкіше, ніж константа
partial	клас	рознести клас у декілька файлів

Конструктори



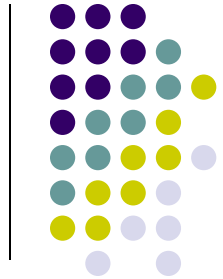
```
class Fraction
{
    private int num;
    private uint den;
// закритий конструктор запобігає створенню екземплярів
    public Fraction() { num = 0; den = 1; }
    public Fraction(int x) { num = x; den = 1; }
    public Fraction(int x, uint y): this(x)
     { if (y > 0) den = y ; }
    public Fraction(Fraction f)      // не властиво C#
     { this.num = f.num; this.den = f.den; }   …
}
    Fraction A = new Fraction();
    Fraction B = new Fraction(1){ den = 2 }; // для відкритих членів
```

Оголошення, використання методу



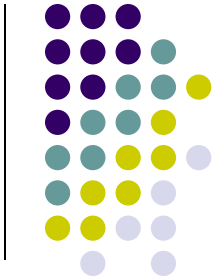
- [модифікатори] тип ім'я ([параметри]) { тіло методу }
- параметри-значення і параметри-посилання (всі ініціалізовані перед передачею):
  - типи-значення передаються за значенням – копіюються
  - типи-посилання передаються за посиланням – копіюється посилання
  - рядки передаються як значення, бо зміна рядка створює новий рядок
  - примусове передавання посилання – префікс ref
  - передавання неініціалозованого параметра за посиланням для отримання результату – префікс out
  - bool MyMethod(int a, ref int b, out int c)
    { ++b; c=a+b; return c>b; }
  - int x=5; int y; bool rez = MyMethod(-2, ref x, out y);
- аргументи можна іменувати (?), тоді їхній порядок не важливий
  - MyMethod(c: y, a: -2, b: x);
- перевантажені методи: однакові імена, різні сигнатури; нема (?) необов’язкових параметрів

Властивості в класі C#



```
class Money
  private decimal amount;  // поле властивості для зберігання значення
  public decimal Amount    // інтерфейс доступу
  {
    get                    // метод читання, тип decimal, без параметрів
    {
      return amount;
    }
    set                    // метод запису з єдиним параметром value
    {                      //   типу decimal
      amount = (value > 0) ? value : 0;
    }
  }                        // кінець оголошення властивості
  public override string ToString()
  {
    return "$" + Amount.ToString();
  }
}
```

Автоматичні властивості та ще дещо

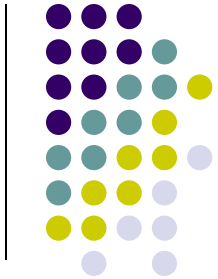


```
class Money
{       // поле властивості для зберігання значення створить компілятор
   public decimal Amount { get; set; }
   public override string ToString()
   {      return "$" + Amount.ToString();     }
}
```

- обидва методи доступу обов’язкові
- один з методів можна зробити закритим чи захищеним
- навіщо потрібні автоматичні властивості:
  - методи доступу можуть мати різну видимість (protected)
  - властивості оголошують “про запас” – завжди можна додати функціональності
  - поле – це дані, властивість – це функція, тому її легше відлагоджувати

- у звичайних властивостей один з методів може бути відсутнім, тоді – лише для читання, або лише для запису (краще зробити метод)

Статичний конструктор, статичний клас

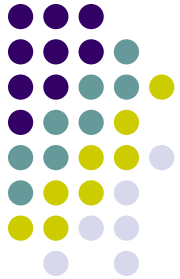


```
using System;
using System.Drawing;
namespace
    Wrox.ProCSharp.StaticConstructorSample
{
  public class UserPreferences
  {
    public static readonly Color BackColor;
    static UserPreferences()
    {
      DateTime now = DateTime.Now;
      if (now.DayOfWeek ==
            DayOfWeek.Saturday
        || now.DayOfWeek ==
            DayOfWeek.Sunday)
        BackColor = Color.Green;
      else
        BackColor = Color.Red;
    }
    private UserPreferences()  {  }
  }
```

```
class MainEntryPoint
  {
    static void Main(string[] args)
    {
      Console.WriteLine(
        "User-preferences: BackColor is: " +
        UserPreferences.BackColor.ToString()
      );
    }
  }
}
```

- поле readonly може задати тільки конструктор
- статичний конструктор не має параметрів, явно його не викликають
- завдання статичного конструктора – ініціалізувати статичні поля
- виклик статичного конструктора – перед першим звертанням до класу
- не можна покладатися на порядок викликів таких конструкторів
- статичний і звичайний не плутаються

Розширення функціональності класу

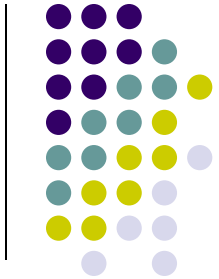


```
namespace Wrox
{  class Money
   {
      public decimal Amount { get; set; }
      public override string ToString()
      {
         return "$" + Amount.ToString();
      }
   }
}
```

```
namespace Wrox
{ public static class MoneyExtension
  {
    public static void AddToAmount(
this Money money, decimal amountToAdd)
    {
      money.Amount += amountToAdd;
    }
  }
}
```

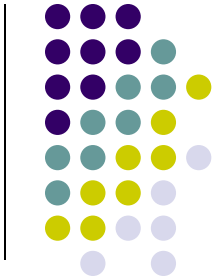
```
class MainEntryPoint
  {
    static void Main(string[] args)
    {
      Money cash = new Money();
      cash.Amount = 40M;
    Console.WriteLine(
         "cash.ToString() returns: " +
         cash.ToString());

    //Extension Method
      cash.AddToAmount(10M);

    Console.WriteLine(
         "cash.ToString() returns: " +
         cash.ToString());
      Console.ReadLine();
    }
  }
```

Методи System.Object



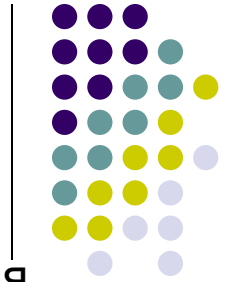
- ToString()
  - віртуальний, повертає назву класу; зазвичай перевизначають
- GetHashCode()
  - перевизначають в класах, чиї екземпляри планують використати як ключі словника
- Equals(), ReferenceEquals()
  - враховують тонкі моменти порівняння об’єктів .NET
- Finalize()
  - використовують для звільнення некерованих ресурсів, працює в ході “збирання сміття” (аналог деструктора)
- GetType()
  - постачає інформацію про клас в екземплярі System.Type
- MemberwiseClone()
  - створює “поверхову” копію отримувача, повертає посилання на неї

Приклад копіювання екземплярів



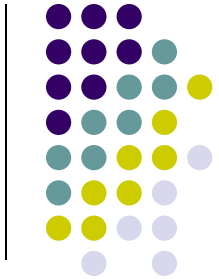
```
public class IdInfo
{
    public int IdNumber;
    public IdInfo(int IdNumber)
    {
        this.IdNumber = IdNumber;
    }
}

public class Person
{
    public int Age;
    public string Name;
    public IdInfo Id;
    public Person ShallowCopy()
    {
      return (Person)
            this.MemberwiseClone();
    }
    public Person DeepCopy()
    {
      Person other = (Person)
            this.MemberwiseClone();
      other.Id = new IdInfo(Id.IdNumber);
      other.Name = String.Copy(Name);
      return other;
    }
}

public class Example
{
    public static void Main()
    {
        Person p1 = new Person();
        Person p2 = p1.ShallowCopy();
        Person p3 = p1.DeepCopy();  …
    }
}
```

Анонімні типи C#



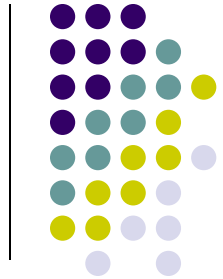
- інкапсуляція в один об’єкт набору властивостей тільки для читання
- тип виводить компілятор за складом ініціалізатора, “придумує” ім’я класу; можна привести до типу object (System.Object)
- var cap = new { FirstName = "James", MiddleName = 'T',
  LastName = "Kirk" };
  var doc = new { FirstName = "Leonard", MiddleName = 'L',
  LastName = "McCoy" };
- cap.GetType().ToString() == <>f__AnonymousType0`3[
  System.String,System.Char,System.String]
- анонімний тип – нащадок System.Object, містить виключно властивості, функціональність виключно успадкована
- типовий приклад використання:
  - Fraction[ ] Rationals = new Fraction[ ]
    { new Fraction(1,2), new Fraction(2,3), new Fraction(3,4)};
    var ratQuery = from R in Rationals select new { R.num };
    foreach (var f in ratQuery) Console.WriteLine("Selected numerator is {0}", f.num);
- анонімний тип не можуть мати поля, події, методи, властивості, конструктори, індексатори
- параметр типу object може прийняти екземпляр анонімного типу

Наслідування



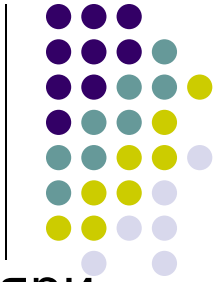
- Наслідування реалізації
  - підклас успадковує набір полів (вкладений об’єкт) і наслідує поведінку (успадковані та перевизначені методи)
  - підклас уточнює, розширяє функціональність базового класу
  - базовий клас реалізує спільну для підкласів функціональність
  - конструктори не наслідуються
- Наслідування інтерфейсу
  - тип наслідує сигнатуру функцій без жодної реалізації
  - угода, контракт на постачання певної функціональності
  - наслідування від класу, що містить лише абстрактні методи
  - наслідування від інтерфейсу (interface)
  - різні класи забезпечують виконання схожих дій (реакцію на однакові повідомлення), але роблять це кожен своїм способом

Синтаксис



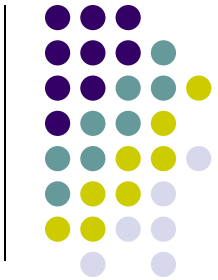
- public class MyClass : object
  { private int a;
    public MyClas(int x) { a = x; }
    public virtual string ToString() { return "BS "+a.ToString(); }
  … }
  public class SubClass : MyClass
  { private int b;
    public SubClass(int x, int y) : base(x) { b = y; }
    public override string ToString() { return "SC : "
        base.ToString() + b.ToString(); }
  … }
- class DerivedClass : BaseClass, IInterface1, IInterface2
  { … }
- public struct record : IInterface1, IInterface2
  { … }

Сумісність і приведення



- У посиланні на базовий клас можна зберігати екземпляри підкласів
  - ```
    MyClass P = new SubClass(30, 45);
    MyClass Q = new MyClass(29);
    object R = new MyClass(72); // неявне приведення
    ```
  - ```
    void DeelWith(MyClass M)
    {   Console.WriteLine(M.ToString());  }
    DeelWith(P); DeelWith(Q); // неявне приведення
    DeelWith( (MyClass) R);     // явне приведення
    ```
- Безпечне приведення
  - ```
    Employee frank = P as Employee;
    if (frank == null) Console.WriteLine("Error with "+P.ToString());
    ```
- Перевірка типу
  - ```
    if (R is MyClass) …
    ```

Інтерфейси



- “контракт” на реалізацію певної функціональності
- може містити оголошення методів, властивостей, індексів і подій
- всі члени інтерфейсу неявно відкриті
- нема конструкторів, перевантажених операцій
- public interface IDisposable
  { void Dispose(); }
- class SomeClass : IDisposable
  {
  public void Dispose()
  { // реалізація }
  ...
  }

Використання інтерфейсів

```
using System;
using Wrox.ProCSharp;
using Wrox.ProCSharp.VenusBank;
using Wrox.ProCSharp.JupiterBank;

namespace Wrox.ProCSharp
{  class MainEntryPoint
   {  static void Main()
      {  IBankAccount venusAccount =
                              new SaverAccount();
         IBankAccount jupiterAccount =
                              new GoldAccount();
         venusAccount.PayIn(200);
         venusAccount.Withdraw(100);
Console.WriteLine(venusAccount.ToString());
         jupiterAccount.PayIn(500);
         jupiterAccount.Withdraw(600);
         jupiterAccount.Withdraw(100);
Console.WriteLine(jupiterAccount.ToString())
         ;
}  }  }
namespace Wrox.ProCSharp
{  public interface IBankAccount
   {  void PayIn(decimal amount);
      bool Withdraw(decimal amount);
      decimal Balance { get; }
} }
```

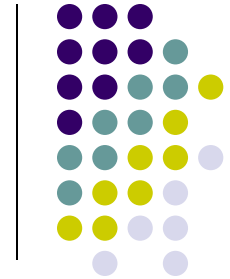
```
namespace Wrox.ProCSharp.VenusBank
{  public class SaverAccount : IBankAccount
   {  private decimal balance;
      public void PayIn(decimal amount)
      {  balance += amount;   }
      public bool Withdraw(decimal amount)
      {  if (balance >= amount)
         {  balance -= amount;
            return true;    }
         Console.WriteLine(
                  "Withdrawal attempt failed.");
         return false;
      }
      public decimal Balance
      {  get  {  return balance;  }  }
      public override string ToString()
      {  return String.Format("Venus Bank
       Saver: Balance = {0,6:C}", balance);
} }  }

namespace Wrox.ProCSharp.JupiterBank
{  public class GoldAccount : IBankAccount
   {  private decimal balance;
      public void PayIn(decimal amount)
      {  ...  }
 ...}
}
```

Наслідування інтерфейсів

```
namespace Wrox.ProCSharp
{  public interface ITransferBankAccount :
       IBankAccount
   {
      bool TransferTo(IBankAccount
       destination, decimal amount);
   }
}

namespace Wrox.ProCSharp
{
  class MainEntryPoint
   {   static void Main()
       {
        IBankAccount venusAccount =
                                  new SaverAccount();
        ITransferBankAccount jupiterAccount =
                                new CurrentAccount();
        venusAccount.PayIn(200);
        jupiterAccount.PayIn(500);
        jupiterAccount.TransferTo(
                                  venusAccount, 100);
Console.WriteLine(venusAccount.ToString());
Console.WriteLine(jupiterAccount.ToString())
       ;}
   }
}
```

```
namespace Wrox.ProCSharp.JupiterI
{  public class CurrentAccount :
       ITransferBankAccount
   {
      private decimal balance;
      public void PayIn(decimal amount)
      {  balance += amount;   }
      public bool Withdraw(decimal amount)
      {  if (balance >= amount)
         {  balance -= amount; return true; }
         Console.WriteLine("Withdrawal attempt
       failed.");
         return false;      }
      public decimal Balance
      { get { return balance; }
      }
      public bool TransferTo(IBankAccount
       destination, decimal amount)
      {  bool result;
         if ((result = Withdraw(amount)) == true)
            destination.PayIn(amount);
         return result;   }
      public override string ToString()
      {  return String.Format("Jupiter Bank
       Current Account: Balance = {0,6:C}",
                                          balance);
} } }
```

І на завершення



- Використання захищених членів класу
- “Запечатані” класи в ієрархії
  - partial class Employee { … }
    public class Manager : Employee { … }
    public class SalesPerson : Employee { … }
    public sealed class PTSalesPerson : SalesPerson { … }
- Відношення “has-a”
  - class BenefitPacadge
    { public double ComputePayDeduction() { … } … }
    partial class Employee
    { protected BenefitPacadge empBenefits;
      public double GetBenefitCost()
      { return empBenefits.ComputePayDeduction(); } … }
- Вкладені типи (для класів і структур)
  - public class OuterClass
    { public class PublicInnerClass { }
      private class PrivateInnerClass { }
    …}